BEHAVIOR FLEXIBILITY FOR AUTONOMOUS UNMANNED AERIAL SYSTEMS

THESIS

Taylor B. Bodin, Second Lieutenant, USAF

AFIT-ENG-MS-18-M-011

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-011

BEHAVIOR FLEXIBLITY
FOR AUTONOMOUS UNMANNED AERIAL SYSTEMS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Taylor B. Bodin, B.S.E.E.

Second Lieutenant, USAF

March 2018

AFIT-ENG-MS-18-M-011

BEHAVIOR FLEXIBLITY

FOR AUTONOMOUS UNMANNED AERIAL SYSTEMS

THESIS

Taylor B. Bodin, B.S.E.E.
Second Lieutenant, USAF

Committee Membership:

Maj Jason M. Bindewald, Ph. D.
Chair

Dr. Gilbert L. Peterson
Member

Dr. Robert C. Leishman
Member

Dr. David R. Jacques
Member

Abstract

Autonomous unmanned aerial systems (UAS) could supplement and eventually subsume a substantial portion of the mission set currently executed by remote pilots, making UAS more robust, responsive, and numerous than permitted by teleoperation alone. Unfortunately, the development of robust autonomous systems is difficult, costly, and time-consuming. Furthermore, the resulting systems often make little reuse of proven software components and offer limited adaptability for new tasks. This work presents a development platform for UAS which promotes behavioral flexibility. The platform incorporates the Unified Behavior Framework (a modular, extensible autonomy framework), the Robotic Operating System (a RSF), and PX4 (an open-source flight controller). Simulation of UBF agents identify a combination of reactive robotic control strategies effective for small-scale navigation tasks by a UAS in the presence of obstacles. Finally, flight tests provide a partial validation of the simulated results. The development platform presented in this work offers robust and responsive behavioral flexibility for UAS agents in simulation and reality.

This work lays the foundation for further development of a unified autonomous UAS platform supporting advanced planning algorithms and inter-agent communication by providing a behavior-flexible framework in which to implement, execute, extend, and reuse behaviors. The contributions within have been presented at the 2017 SOCHE STEM-Cyber Research Symposium, the 2017 ION Joint Navigation Conference, and the 2018 SSC Pacific Naval Applications of Machine Learning Conference.

Acknowledgements

I would like to offer my deepest appreciation to my advisors, both present and past, for their guidance, encouragement, and patience. And to my wife, your love and support sustained me through this journey. I couldn't have done it without you. Thank you.

Taylor B. Bodin

Table of Contents

Page

Page

List of Figures

Figure Page

List of Tables

BEHAVIOR FLEXIBLITY

FOR AUTONOMOUS UNMANNED AERIAL SYSTEMS

I. Introduction

Autonomous vehicles represent one of the most disruptive technological innovations in modern warfare [4, 5]. Military applications for autonomous vehicles are numerous and varied; encompassing intelligence, surveillance, reconnaissance, search and rescue, command and control, cargo and munition delivery, and even the munitions themselves [4]. Autonomous vehicles are a significant opportunity for the Department of Defense to reduce man-hours, cost, and risk to human operators. As with all disruptive technologies, however, autonomous vehicles also pose a significant risk to large, successful, and entrenched militaries [5]. These vehicles are becoming more intelligent and capable while simultaneously becoming relatively inexpensive to mass produce, placing them within the reach of nations and many non-state actors. Swarms of intelligent, weaponized vehicles hold cornerstone systems vital to United States military supremacy at risk. Defending a lumbering aircraft carrier, an air base, or even a satellite constellation against an onslaught of hundreds or thousands of small, intelligent drones is a daunting technical challenge that must be addressed in the coming decades [5].

While few operational examples of autonomous, military vehicles exist today, adversaries are aggressively pursuing policies to gain the advantage in this new domain. China has issued the Next Generation Artificial Intelligence Development Plan [6] with the goal of "becoming a global innovation center" by 2030. Specifically, the plan details China's intent to invest billions of dollars in AI research and development, the

emerging AI economy, and products using AI technology including "smart vehicles". Russia is also aware of the huge strategic potential of AI. In an address to thousands of Russian students, Vladimir Putin said, "Whoever becomes the leader in [AI] will become ruler of the world" [7]. While such remarks might seems exaggerated, the disruptive effect of autonomy is evident in the private sector with the rise of tech giants like Amazon, Google, and Facebook, who are developing and employing autonomous systems to tremendous effect [8, 9].

Given the disruptive potential of autonomous vehicles and world-wide interest in their military applications, it is of the utmost strategic importance that the United States and its allies lead in the research, development, and adoption of autonomous systems. Given its recent history of technological supremacy, it may seem that the U.S. Department of Defense is well positioned to maintain its preeminence. However, the current acquisitions process is too costly and slow to keep pace with the rapid progress being made in the field [10, 11, 12]. For example, the Long Range Anti-Ship Missile (LRASM) is a prime case study for the noncompetitive autonomous systems acquisitions process in action. The system itself is an extremely capable autonomous munition, capable of navigating in GPS denied environments, discriminating targets from non-targets, evasive maneuver, and targeted strikes [13]. However, its "accelerated acquisition" is still in progress after 7 years and an estimated program cost of 1 billion dollars for a mere 110 units [14]. If these metrics are indicative of future procurement programs, they set a low bar for adversaries to meet or exceed.

1.1 Research Motivation

To keep pace with adversaries, the procurement and employment of future autonomous vehicles must be more efficient. While many factors influencing the acquisition and life cycle management of autonomous systems are political in nature,

the modularity and behavior flexibility of a platform are key technical characteristics which contribute to efficiency in employment. Modularity describes the ease with which system components (both in hardware and software) can be extended, modified, reused, and interchanged. behavior flexibility regards the number of heterogeneous behavior the platform can competently perform. Behaviors can be described in terms of their: (1) robustness, the degree to which the behavior can cope with uncertainty in *a task*; and (2) adaptability, the ease with which the behaviors can be extended or reused to perform *new tasks*. The LRASM for instance would likely score low in modularity and adaptability, since the software is closed-source and highly optimized for its domain, but score well in robustness given its ability to perform despite a high degree of uncertainty in its task. If systems are designed to be modular and behaviorally-flexible, development cost per task drops exponentially as the number of tasks the system can accomplish increases.

With respect to physical vehicles, unmanned aerial systems (UAS) are effective, behavior-flexible platforms due to their high mobility in open environments. A multicopter equipped with a high resolution camera, for instance, can perform a large, and growing, set of tasks, including 3D mapping, reconnaissance, surveying, and aerial photography. To date, many applications of UAS rely on teleoperation or basic automation [15]. The creation of autonomous UAS has proven to be an extremely demanding engineering challenge and is currently an area of active research. Solutions must fuse sensor inputs from heterogeneous, noisy, and often unreliable physical devices; reason about the state of dynamic and ill-defined environments; plan actions; and execute those plans on hardware. Due to the nature of flying systems, the above must be accomplished in near real-time and within restrictive size, weight, and power requirements. While many exciting and capable applications of artificial intelligence are flying on research UAS today, progress has been hindered by high levels of dupli-

cation of effort [16, 17, 18]. Great strides have been made in the creation of capable aerial robotics platforms [19, 20, 21] which allow users to quickly develop and deploy new robotic hardware, but autonomous UAS are lacking an analogous platform allowing the development and reuse of intelligent agents.

The Unified Behavior Framework (UBF)[1] provides a design strategy to create modular and extensible behavior-based robotic agents. For many years, behavior-based approaches have produced robust and responsive intelligent agents. However, in robotic applications, behavior logic is often inextricably tied to the underlying control mechanisms making reuse, modification, and extension of behaviors difficult. The UBF was developed to address these limitations by abstracting behavior logic from the underlying robotic controllers. Behavior abstraction allows developers to easily reuse and modify behaviors to extend an existing behavior-based controller or to quickly create a new controller. The efficacy of the UBF was demonstrated through successful implementation on a variety of platforms [22, 23, 1]. However, overall agent behavior-flexibility was limited by their ground-based platforms. Open-source robotic software frameworks (RSF), such as the Robot Operating System (ROS) [21] and flight controllers such as the PX4 autopilot [20], currently offer a rich application programming interface (API) for UAS applications that could facilitate the integration of UBF behavior logic with a variety of sensors and physical vehicle types [24, 25, 26]. In addition to being the first use of the UBF on a UAS before, the integration of these technologies offers tremendous potential as behavior-flexible platform for autonomous agents on UAS.

1.2 Research Questions and Hypothesis

The objective of this research is to answer the following question, "Is it possible to develop a behavior-flexible development platform for autonomous UAS agents using

open-source software components?" It is hypothesized that if the UBF is integrated with a suitable robotic software framework (RSF), then the resulting framework produces agents that are viable for real-world use, robust to uncertainty in the environment, and adaptable to different behavior. Five investigative questions (IQs) were devised to test this hypothesis and are listed below.

- IQ1 - What is the current state-of-the-art with regard to RSFs and open-source flight controllers for small UAS? A wide variety of RSFs and flight controllers are currently available. A survey of these components is intended to identify state-of-the-art components suitable for a behavior-flexible UAS development platform.

- IQ2 - What design produces a behavior-flexible development platform? Based on the literature review, a design is proposed which integrates the UBF with the Robotic Operating System (ROS), a powerful RSF, and PX4, a leading flight controller. This setup provides 2 benefits. First, the addition of new sensors and actuators is simplified by the ROS communication middleware. Second, the platform is portable to many different UAS types, including fixed-wing, multi-rotor, and vertical takeoff and landing (VTOL).

- IQ3 - Which reactive robotic paradigm, or combination of paradigms, produces behavior-flexible agents with the highest fitness on navigation tasks? The development platform supports agents based on novel combinations of traditional reactive robotic paradigms. As the first application of the UBF on UAS, an assessment of agent fitness, robustness, and adaptability is intended to identify effective, behavior-flexible control strategies for autonomous UAS.

- IQ4 - Is the system safe, stable, and compliant with testing regulations? Vehicles using the development platform will not be authorized to operate without

demonstrating regulatory compliance and basic functionality. A flight test program using the build-up approach is necessary to safely demonstrate agents are functional and compliant.

- IQ5 - To what degree does simulated agent behavior predict actual agent behavior? Simulation provides a safe, low-cost, and convenient environment to validate and tune agent behavior. The usefulness of simulated results depends on how closely simulation reflects reality.

By answering the investigative questions outlined above, this research will be able to design and validate a behavior-flexible development platform comprised of state-of-the-art components. Using the platform, an assessment of UAS-based agent performance on navigation tasks will yield important insights for the design of effective reactive-robotic control architectures.

1.3 Limitations and Assumptions

In order to form a valid logical argument answering the research question, several simplifying assumptions were made. The first assumption is that RSFs and flight controllers can be objectively ranked to make design decisions. There is no objective basis on which to rank RSFs or flight controllers which complicates establishing the state-of-the-art. For this research, a comparison framework from the literature was used to compare actively developed projects. It is a assumed that the RSF with the most desirable proprieties for the development platform constitutes the state-of-the-art. Design considerations were made on the basis of this survey and the particular needs of the development platform.

Second, this research was inherently limited in scope by the size of the design and test spaces. The test space was reduced to three navigation tasks designed to reflect

common features of many navigation tasks. It is assumed that results from these test cases generalize to similar tasks. The design space was reduced by only manipulating a focused subset of the design parameters. It is assumed that the interaction effects between constant parameters and experimental parameters are relatively small. If this assumption is true, then agent fitness is mostly determined by the experimental factors rather than any particular set of constant parameters.

Third, it is infeasible to fully validate the safety, stability, and compliance of the platform because of the nature of emergence and the limited number of flight test sorties. It is assumed that the risk posed by unexpected or unstable behavior is manageable and that a partial validation is sufficient to demonstrate that the platform is viable in an experimental flight test setting.

Finally, several assumptions were made about the portability of the development platform. It is assumed that no modifications would be necessary if porting to another multirotor type UAS if: (1) behaviors operate within the flight envelope of the target vehicle and (2) the flight controller is properly tuned for the target vehicle. It is also assumed that little modification would be needed to port the platform to other UAS. This assumption relies on the fact that the PX4 flight controller can effectively control multiple vehicle types using the same set of commands.

1.4 Thesis Organization

The following chapters document the verification and validation of the proposed platform. Chapter II provides background and context for the problem. Chapter II begins with a synopsis of the history of autonomous robotics and a discussion of the UBF. Next, the general characteristics of RSFs are discussed along with a comparison of RSFs under active development. This comparison is used to assess the potential behavior-flexibility offered by each RSF to provide justification for design decisions.

Chapter II concludes with an overview of behavior based robotics, the UBF, and related work utilizing the UBF. Chapter III describes the implementation and test methodology for the proposed architecture. The chapter begins by detailing the hardware and software implementation of the platform. A methodology is proposed which attempts to determine if the system is: (1) compliant with testing regulations; (2) robust to dynamic and unstructured tests; and (3) adaptable to new behaviors. An exploration of the effect of parameter variation on agent performance is conducted to determine configurations that are well suited to this new domain for the UBF. Chapter IV presents the results and analysis of the simulated and flight test experiments with a discussion of their significance. Finally, Chapter V summarizes the findings of this research effort and provides avenues for future research and development for the platform.

II. Background

The creation of intelligent, flying robots is a uniquely challenging engineering problem. For an autonomous robotic agent to reliably accomplish tasks it must, at a minimum, fuse sensor input from a number of heterogeneous, noisy, and often unreliable physical devices, use that data to make sense of its environment, plan actions, and execute those plans on hardware [27]. This challenge is even greater for UAS due to the nature of flying systems. High speeds, environmental variation such as weather, and relatively large operating areas create a dynamic and unpredictable operating environment. To perform safely and rationally, the vehicle must accomplish most of the above tasks in near real-time and within restrictive size, weight, and power requirements. Despite these challenges, advancements in battery energy density and processing capability have precipitated rapid growth in SUAS commercial products, technology, and research in recent years [28]. As the complexity and number of applications for these systems grow, the need for a behavior-flexible development platform for UAS is increasingly apparent [29, 4].

Such a behavior-flexible development platform must include three fundamental components of an autonomous UAS: a flexible autonomy framework, a flight controller, and a robotic software framework (RSF) to tie the two together. The purpose of this chapter is to provide the reader with sufficient background to understand the role of each component and resolve IQ1 by surveying the candidate components to establish the current state-of-the-art. This chapter opens with a brief review of autonomous robotic paradigms, including deliberative, behavior-based, and three-layer architectures. Next, the Unified Behavior Framework is presented as a behavior-based autonomy framework which is well suited for use in UAS. Then, a short primer on the control of multi-rotor UAS is presented and two prominent flight controllers are introduced. Next, a discussion of the general features of RSFs provides context for a

survey of several RSFs currently under active development. At the end of the chapter, the selected components are presented with justification for their inclusion in the final architecture.

2.1 Autonomous Robotic Paradigms

This section provides a brief history and introduction of the fundamental paradigms of autonomous robotics. The first section overviews the hierarchical paradigm, a top-down approach to artificial intelligence. Next, is the reactive paradigm is described as a response to limitations of a deliberative approach. The UBF is then introduced as a behavior-flexible and extensible framework with which to create reactive robotic agents. Work related to the UBF, including its implementation and simulation in multiple environments, is also discussed. The section concludes with an overview hybrid architectures which combined elements of the aforementioned paradigms and are currently the state of the art in autonomous robotics.

2.1.1 Hierarchical Paradigm.

The hierarchical or deliberative approach to autonomous control is the oldest of the robotic paradigms and revolves around three primitives: sense, plan, and act [27]. First, the robot perceives the world through its sensors and creates a representation of its environment called the world model. The robot then uses the world model to plan a series of actions to get from its current state to a goal state. Finally, plans are carried out in the act phase and begins the cycle again from the sense phase.

While the hierarchical paradigm is highly intuitive, it suffers from two major shortcomings. First, in a purely hierarchical architecture, the robot is unresponsive to the environment during the planning and action phase. Since perception and planning are typically computationally complex and non-deterministic in time, robots using this

approach are unresponsive to dynamic environments. Second, a top-down, symbolic approach to artificial intelligence suffers from the frame problem. For the world model to be useful, it needs to include all information relevant to the robot. First described in [30], the frame problem describes the irreducible complexity of trying to build such a world model. As the environment becomes more complex, the number of axioms needed to sufficiently describe that environment grows so quickly that operating in a realistic environment is infeasible [27]. Figure 1 depicts the arrangement of the sense, plan, and act paradigms within a deliberative controller.

Figure 1. The flow of information through the deliberative paradigm. Information is first sensed. Then a plan is created based on an environmental representation created from sensed data. Finally, the plan is carried out in the act phase.

2.1.2 Reactive Paradigm.

In response to the inherent limitations of the hierarchical paradigm, researchers began pursuing bottom-up approaches to robotic autonomy. Braitenberg, in one of the foundational papers of reactive robotics, proposes a set of imaginary robots called vehicles. Braitenberg's vehicles [31] are devoid of any form of environmental representation and instead tie sensory input directly to motor outputs. Through a series of vehicles of increasing complexity, Braitenberg illustrates that very complex behaviors can emerge from stateless structures relying on simple mechanisms. Figure 2 depicts this structure consisting of sensing linked to action through behaviors.

Brooks [32] created autonomous robots with control structures similar to those imagined by Braitenberg using the subsumption architecture. The subsumption architecture turns the horizontal, i.e. sequential, decomposition of functionality in

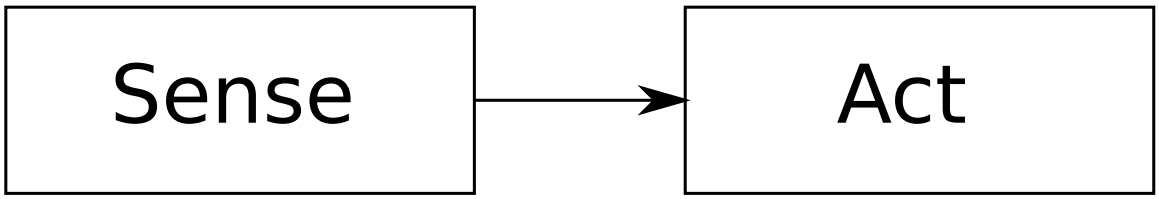

Figure 2. The flow of information through the reactive paradigm. Notice the absence of the planning stage from the hierarchical paradigm. Sensed information is converted into action with little to no state representation.

the hierarchical paradigm on its side. The resultant structure produces a vertical hierarchy in which behaviors are capable of concurrent operation. Each behavior is organized according to their level of competence in layers with low-level behaviors handling "survival functions" and higher levels addressing the robot's goals [27]. Each behavior suggests actions concurrently with higher level behaviors *subsuming* lower levels in the event of a conflict. Subsumption was successfully demonstrated on a number of small, insect-like robots which were revolutionarily responsive and agile for their time [32].

Due to the success of these new methodologies, researchers began exploring new areas of behavior-based robotics in the late 1980's and early 1990's. One particularly useful class of reactive architecture, especially for navigation based tasks, that emerged from this era are potential field methodologies. First introduced by Ronald Arkin [33], potential field methodologies work by modeling behaviors as a potential field indicating the desired motor action at any point in the field. Whereas subsumption achieved emergence by layering behaviors according to their priority, potential field methodologies sum the fields produced by each behavior producing a complex net potential field. Behaviors are defined according to a set a primitive field topologies (i.e. uniform, tangential, radial, etc.) and magnitude profiles (i.e. constant, linear, exponential, etc.). These methods have proven capable in real-world application, easy to visualize, and straightforward to implement [27].

2.1.3 The Unified Behavior Framework.

Traditionally, behavior-based architectures like subsumption and potential field methodologies are designed to operate within a narrow ecological niche and on one platform. Once tuned for their environment, these controllers can achieve very good results but are limited to the strengths of the chosen architecture and are difficult to extend to additional functionality. Furthermore, as the controller competency grows to accommodate more functionality, the complexity of the control quickly reaches a capability ceiling since traditional architectures lack a mechanism for managing complexity [34].

To address these limitations, Woolley, et. al. [34] introduced the UBF for the construction of reactive controllers which are extensible, architecture agnostic, and manage the complexity of highly competent controllers. The UBF is a software engineering pattern which applies the composite and strategy pattern [35] to the development of behavior-based controllers. Using the strategy pattern [35], the UBF abstracts behavior logic from the underlying robotic controller by encapsulating it into a standardized interface. Since the controller knows how to use this abstraction, any concrete implementation, regardless of its reactive architecture, is viable. This implies that not only can disparate architectures be used within the same architectures, but they can be swapped out at runtime as necessary, thus freeing the controller from the strengths or weaknesses of any one reactive architecture.

By using the composite pattern [35], the UBF allows designers to create complex, mediated hierarchies of behaviors. Two or more behaviors can be combined into a composite behavior as depicted in the class diagram in Figure 3. Each child behavior suggests an action based on the current state. These action recommendations form **A**, the set of proposed actions. An arbiter, another flexible component in the UBF, uses some arbitration mechanism to generate **a'**, a single action, from **A**. Then **a'** can

then be used in another composite behavior to form another layer in the hierarchy if desired. This design pattern promotes reuse of subcomponents in the hierarchy and provides a mechanism for managing the complexity of sophisticated controllers [34].

2.1.4 Related Work.

The UBF has been previously demonstrated in several robotics applications. Woolley in [34] introduces the UBF and conducts several case studies to exhibit the capabilities of the framework. In the first two case studies, Woolley simulated UBF agent controlled battle tanks in the Robocode robot battle environment. These experiments highlighted the flexibility of UBF agents through structural variation of the behaviors and arbitration elements. In his final case-study, Woolley implemented the UBF on the Pioneer P2-AT8, a four-wheeled, mobile robot with sonar, LIDAR, bump, and odometery sensors. To show the real-world viability of the UBF with hard real-time constraint, the agent was sent high amounts of network traffic to simulate high computational loads. Although the agent was not able to reliably maintain real-time control, Woolley demonstrated that UBF controllers can execute at predictable intervals given the low observed jitter for each behavior. In [36], Hooper applied the UBF to the RoboCup Soccer Simulator, a multi-robot testing platform used in the RoboCup robotics competition. Although Hooper did not apply the UBF to a real-world platform, the ability of a team of agents utilizing the UBF to competitively play soccer demonstrates the ability of the UBF to integrate into hybrid architectures and handle complex tasks. In [37], Duffy reinforced this point by integrating the UBF into a hybrid architecture. Three case study experiments were carried out in the Stage simulation environment using a P2-AT8 robot in multiple configurations. In [22], Lin implemented the UBF on the small Mini-WHEGS^{TM}robot. This application of the UBF demonstrates the viability of the UBF in a deeply embedded, resource-

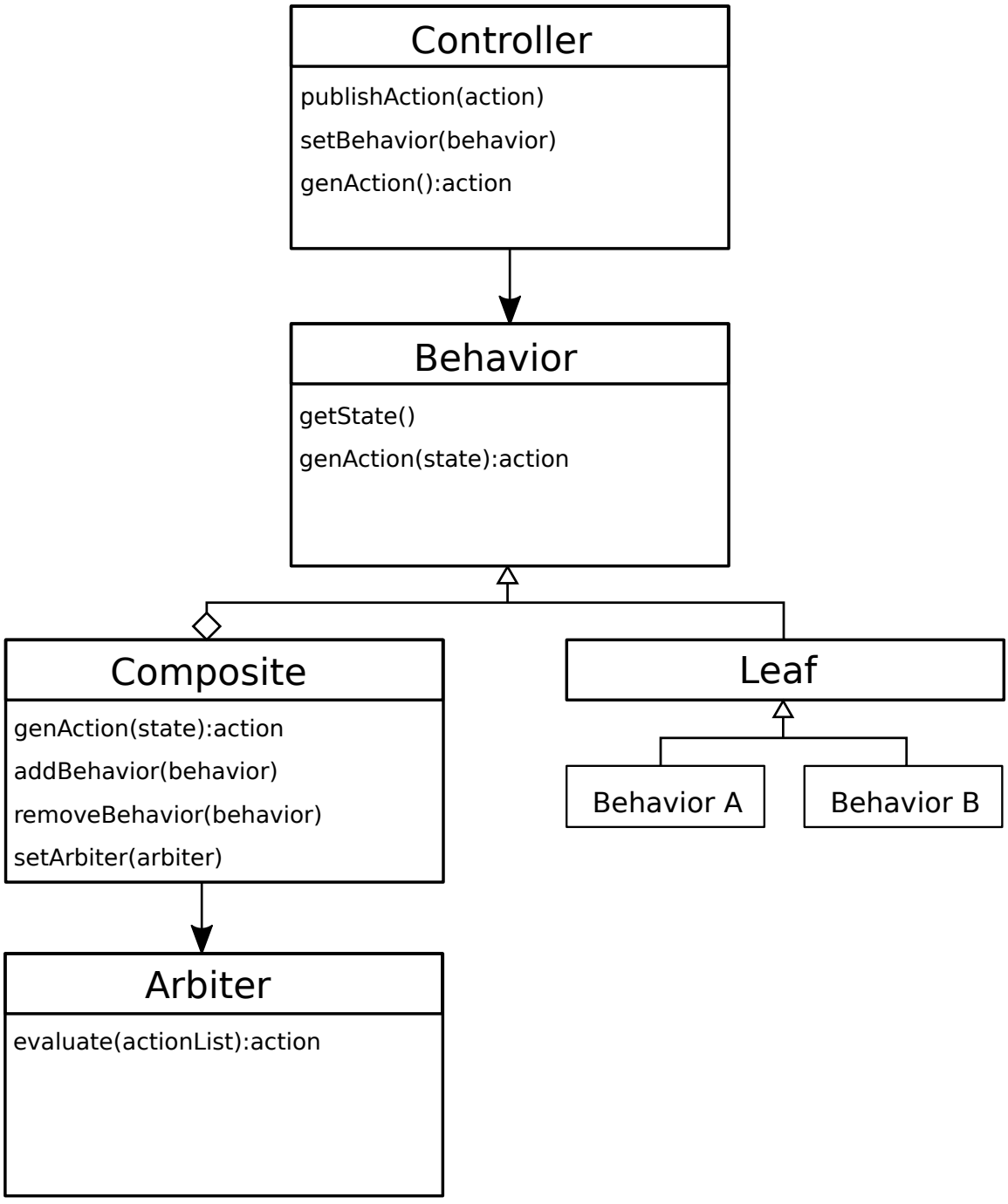

Figure 3. UML class diagram of the Unified Behavior Framework [1].

constrained environment. Finally, Roberson [23] simulated teams of fighter aircraft in multiple mission environments using an architecture based on the UBF. This work also demonstrates the usefulness of UBF based agents for multi-robot applications. A summary of these related works is presented in Table 1.

Table 1. Previous applications of the UBF related to this work.

Author	Simulation	Platform	RSF
Woolley[34]	Robocode	Pioneer P2-AT8	Player
Hooper[36]	RCSS	Multiple	None
Duffy[37]	Stage	Multiple	None
Lin[22]	None	Mini-WHEGS™	None
Roberson[23]	OpenEaagles	A2A Sims	None

2.1.5 Hybrid Deliberative/Reactive Paradigm.

Although reactive architectures proved to be extremely capable in the early years of autonomous robotics, the reactive paradigm as a whole is fundamentally limited by its inability to remember past events or reason about the future states of the environment [27]. For an autonomous system to avoid local minima, correct for degraded performance, and generate plans, some elements of deliberative architectures needed to be re-introduced. Thanks to increased use of concurrent processing techniques and modular reactive controllers, hybrid architectures could be constructed which had the responsiveness of pure reactive approaches and the long-term goal seeking and error correcting capability found in hierarchical approaches [27]. These hybrid architectures typically consist of three layers: a reactive controller granting responsiveness, a planning layer called the deliberator, and an intermediary layer called the sequencer [34]. The deliberative layer is responsible for maintaining the world model and generating plans. Plans are interpreted by the sequencer which generates a *sequence* of behaviors suitable for accomplishing the plan. Finally, the reactive controller executes this sequence. The hybrid deliberative/reactive paradigm represents the current dominant

paradigm in the field of autonomous robotics. By combining reactive and deliberative approaches, hybrid architectures benefit from the strengths of both allowing for responsive, goal-seeking autonomous robots. Figure 4 depicts the conceptual structure of the sense, plan, and act paradigms within a hybrid controller.

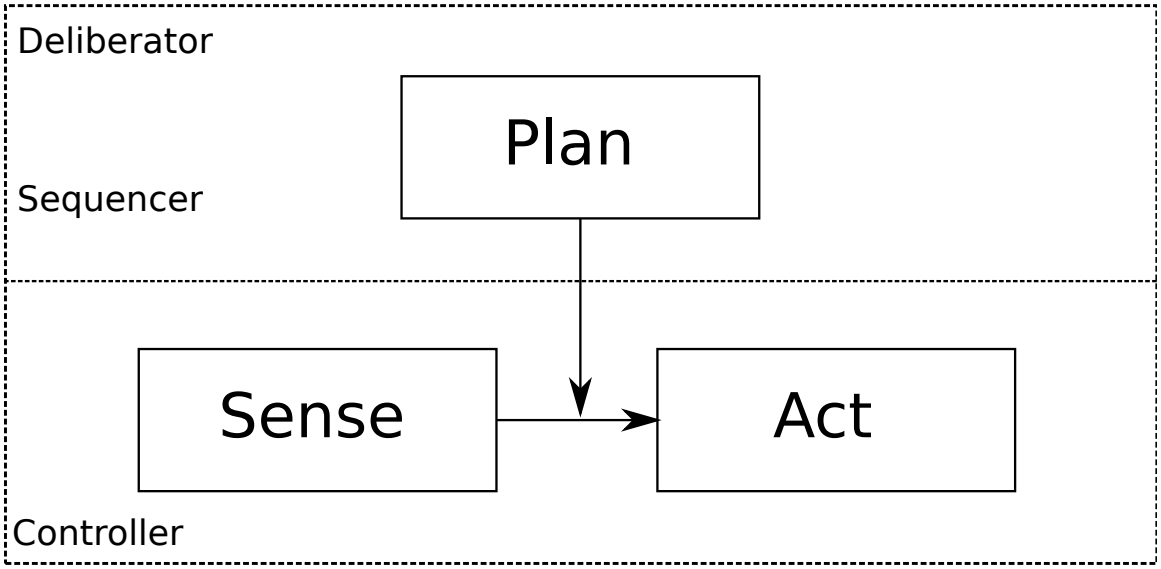

Figure 4. The flow of information through the hybrid deliberative/reactive paradigm. This paradigm possesses elements of both deliberative and reactive paradigms. Plans are created from a environmental representation and then executed through a sequence of behaviors.

2.1.6 Summary.

The Unified Behavior Framework is a flexible, extensible, and modular design pattern to implement behavioral controllers. The UBF is a design framework that applies common software engineering approaches to the development of behavior-based controllers. Its chief contribution is to abstract behavior logic from actuator controls and encapsulates them with a standardized interface. The modular behavior elements can then be organized according to whatever behavior-based architecture meets the needs of the designer best. Whereas traditional controllers typically only use one architecture and one arbitration mechanism to decide on which recommended

action to take, a UBF based controller can utilize disparate architectural paradigms and different arbitration mechanisms in the same hierarchy. In this hierarchy, lower level behaviors send up action recommendations to an arbiter. These behaviors, called composite behaviors, can then also be combined as desired to produce complex emergent behavior and high levels of skill competency while managing the complexity of the controller. While the UBF has been successfully demonstrated on several mobile robots [22, 34, 36, 37], it has yet to be integrated within a larger RSF and has not flown on a UAS. In general, behavior-based approaches have been shown to be effective in producing the timely and robust responses necessary for the safe operation of autonomous UAS. The UBF increases the flexibility of behavior-based approaches by allowing developers to experiment, extend, and reuse their controllers aboard different platforms. The following section introduces flight controllers which are capable of executing UBF behaviors on a multirotor UAS.

2.2 Control of Multirotor Aerial Vehicles

Flight controllers, also commonly referred to as autopilots or flight management units (FMU), are complex avionics systems which automate the control of UAS. For autonomous UAS, flight controllers form the crucial link between software and hardware by interpreting and executing agent actions. Flight controllers also estimate the aircraft state, which consists of position, attitude, linear velocity, and angular velocity, using accelerometer, gyroscopes, GPS, barometric, vision, and other sensor data [38]. This section briefly describes the dynamics and control of multirotor UAS and introduces two popular, open-source flight controller projects which constitute the state-of-the-art in this area.

2.2.1 Multirotor Dynamics.

Figure 5 displays the standard reference frames used to define the state of a multirotor aircraft..

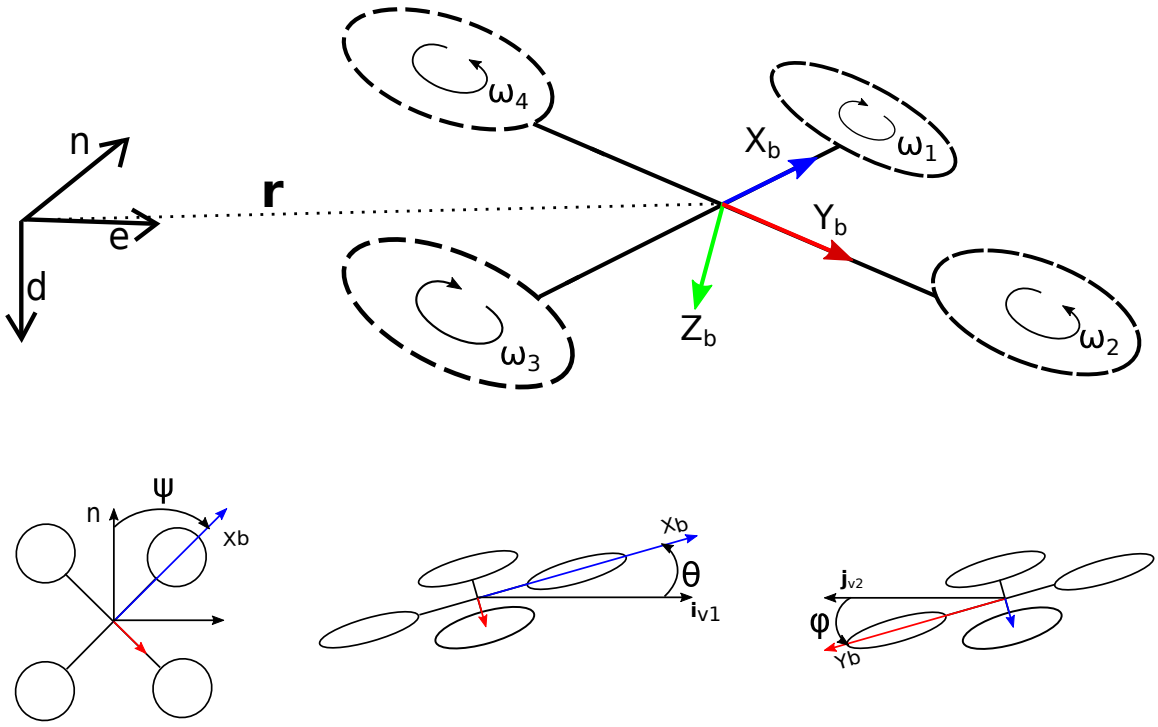

Figure 5. The standard reference frames and axis for multirotor UAS dynamics. The body frame (right) is a transformation r from the inertial NED frame (left). Positive rotations are shown for pitch (θ), roll (ϕ), and yaw (ψ) [2, 3].

The flight controller calculates control inputs to achieve a desired vehicle state also known as a setpoint. In multirotor UAS, control is achieved through the differential thrust of each rotor. Designing robust control laws is difficult because multirotors are under-actuated systems, meaning there are fewer control inputs than outputs. Additionally sensor data is noisy, and the aerodynamic models for multirotor vehicles are only approximate [38, 2]. The most common solution to this problem is to employ a nested loop structure like the one depicted in Figure 6.

A nested control loop structure simplifies the problem of controller design by

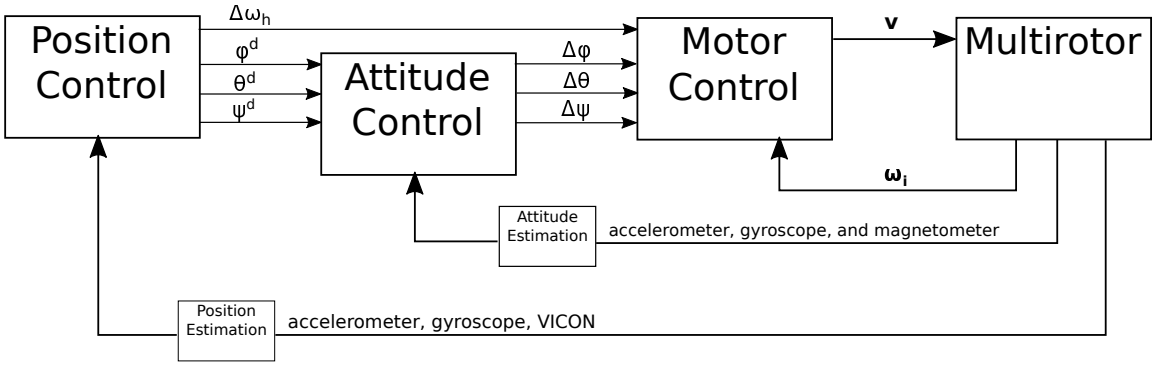

Figure 6. Typical control loop structure for a multirotor UAS flight controller [2, 3].

subdividing the problem into a series of smaller control problems, usually consisting of position control, attitude control, and finally motor control [38]. The position controller sets the desired angular positions: θ^d desired pitch angle, ϕ desired roll angle, and ψ desired yaw angle. The position controller also sets $\Delta\omega_h$, the desired rotation speed for all motors, which roughly corresponds to the desired net thrust. The attitude controller sets the desired angular rates based on the angular position setpoints mentioned above. These setpoints are $\Delta\theta$, $\Delta\phi$, and $\Delta\psi$, which are angular pitch rate, angular roll rate, and angular yaw rate respectively. The motor control loop receives the desired angular rates and gross motor speed and calculates the appropriate voltage for each motor to achieve the desired individual motor speeds. The motor control loop receives feedback in the form of $\omega_\mathbf{i}$ which is the individual motor speeds as reported by the electronic speed controllers attached to each motor.

Designers are able to decompose the multirotor control problem in this way by applying successive loop closure. Successive loop closure allows designers to approximate the inner loops of Figure 6 as unity gain. This is based on the assumption that inner loops operate at a much higher frequency than outer loops, approximately 5 to 10 times higher, so that the transfer function of the inner loop, as seen by the outer loop is essentially flat [3]. The complex feedback structure collapses leaving only an

open loop transfer function consisting of a cascade of the transfer functions of the three individual controllers.

Using this simplification, designers are able to construct three separate controllers that are decoupled from one another. According to [3], the resulting plant models are well controlled by simple proportional-integral-derivative (PID) controllers since they only exhibit first or second-order behavior. Additionally, the overall transfer function of these systems is adequately modeled by second order dynamics since inner loops appear as unity gain to the outer loops. Therefore elementary Linear Time Invariant (LTI) system analysis for second order systems is a useful approach for characterizing flight controllers. For this research, the second order dynamics of interest are:

- Rise time - Time for the output to rise from 10% to 90% of the final output value.
- Latency - time from the start of the input to 10% of the final output.
- Frequency Cutoff - the frequency where output magnitude is 3 dB below the passband.

These measures were chosen because they capture the responsiveness of the controller. Other second order dynamics, such as settling time, overshoot, and oscillatory behavior were disregarded since they are largely determined by tuning of the flight controller itself which was not the subject of this research.

2.2.2 PX4 and Ardupilot Firmware.

There are several capable flight controller firmwares which use some variation or combination of PID controllers arranged in a nested loop structure. A comprehensive survey of open-source flight controller projects is presented in [29]. PX4 and Ardupilot are the only projects which are still being actively developed from this survey.

Interestingly, the two projects were jointly developed in a collaborative project until breaking changes were made due to political reasons in 2016 [39]. It is difficult to select one flight controller over the other on the basis of performance since they were co-developed. Many of the underlying algorithms are very similar and no performance comparison exists in the literature. In terms of behavioral-flexibility, PX4 benefits from a highly modular code base facilitated by node modules communicating over the μOrb publish and subscribe middleware. This is in contrast to Ardupilot firmware which is hindered by a monolithic control loop structure[20, 19, 40]. For this reason, PX4 was selected as the flight controller firmware of interest for this project and likely constitutes the state-of-the-art for open-source flight controllers.

2.3 Robotic Software Frameworks

The development of robotic agents is a full spectrum engineering challenge spanning from low-level hardware and physical device concerns, such as sensors and actuators, to the high-level, almost philosophical, concerns of autonomy and artificial intelligence. Robotic software frameworks (RSFs) were developed in an attempt to bridge this chasm of engineering by providing the tools and infrastructure necessary for robotic software. For this research, an RSF was needed to connect the sensors to the UBF and the UBF to the controllers in a modular and extensible way. RSFs also provided a number of important ancillary functions such as data logging, system introspection tools, and development tools. The following subsections highlight the features and capabilities of RSFs by surveying popular RSFs under active development at the time of writing. This section concludes with a justification of the RSFs chosen for the development platform.

2.3.1 Features of Robotic Software Frameworks.

Precisely defining the capabilities and features of RSFs as a whole is difficult since individual frameworks are typically created with specific design principles in mind and then tailored to a narrow set of applications and/or platforms. As a result of this specialization, RSFs exhibit great diversity with regard to implementation, capabilities, and design philosophy. Currently, there are over 22 well-known, open-source frameworks available. Several authors [24, 41, 25, 26] have attempted to organize and compare these RSFs using an objective basis of comparison. The foundational paper [25] in the area of RSF comparison created a broad conceptual framework for comparing RSFs along four dimensions: specification, platform support, infrastructure, and implementation. The authors then survey nine popular RSFs available in 2007. Due to the extremely high rate of turnover in the last decade, none of the original RSFs analyzed are being actively developed, i.e. major updates being published in the last year. Although it was not the first survey of RSFs [42, 43], the Kramer framework represents one of the most cited and comprehensive in the literature by identifying the major features and themes of RSFs.

The specification category concerns the design principles of an RSF with regard to its incorporation of autonomy paradigms and software engineering. The specification category is divided into three major subcategories: architectural primitives, software engineering, and architecture neutrality. Architectural primitives regard the low level "functional components and/or knowledge primitives" of the system. These primitives form the language of the RSF and can strongly influence the types of control and applicable to the system. In practice, Kramer, et al. [25] quantify an RSF's level of support by the number or form of robot control offered. RSFs with one form of robot control, such as a finite state machine (FSM) or a behavior-based architecture, to receive a somewhat supported rating, and RSFs with more than one

or complex forms of control receive a well-supported rating. Architectural neutrality is also related to the ability of the RSF to accommodate various architectures. RSFs may be strongly associated with a specific agent architecture or completely neutral, leaving the choice up to the developer. Overall, the specification category attempts to address the flexibility afforded by an RSF at design time.

Platform support addresses operating systems, sensors, and effectors supported by an RSF. Platform support has the biggest impact on the ability of software written in an RSF to be reused from one robot to another. The author also included simulation capability in this category. Simulators are, themselves, a kind of invaluable platform allowing developers to test software quickly and with little risk. Finally, the configuration method of the robot falls within platform support. This factor rates the ease with which operational parameters can be modified within an RSF. A difficult configuration method may require a developer to recompile code, while an easy configuration method might allow a user to change parameters at runtime through a GUI. The platform support category addresses aspects of RSFs that affect the execution of robotic software.

Infrastructure encompasses the aspects of an RSF that support the execution of software regardless of the specific architecture utilized or other system-wide design considerations. Kramer et al. lists nine considerations for evaluating an RSFs infrastructure: low-level communication, logging facilities, debugging facilities, distribution mechanisms, scalability, component mobility, system monitor/management, security, and fault-tolerance. The purpose of many categories is self-evident so only a few of the less obvious features will be expounded on here. Distribution mechanisms refer to the ability of an RSFs, usually through networking middleware, to operate in distributed systems such as a swarm. Component mobility describes the ability to "relocate components at run-time".

Finally, implementation pertains to characteristics of the actual implementation of robotic software within an RSF, as well as the "predefined components" available to a developer. Implementation characteristics of interests are: programming language support, high-level robotics languages, documentation, real-time operation, graphical interface, and software integration. In particular, the inclusion of documentation and a graphical interface greatly enhance a developer's ability to write and debug robotic software.

Since 2007, many RSFs were developed. The following sections briefly describe a selection of active, mature RSFs suitable for use in SUAS. Following this overview, the Kramer conceptual framework is applied to each RSF based on information from their respective documentation as well as survey papers which concern RSFs [24, 41, 25, 26]. The following analysis of the state-of-the-art RSFs using the Kramer framework is intended to illustrate the essential elements of RSFs in general while also evaluating the specific RSFs surveyed. A short description of the RSFs to be analyzed is presented in Table 2.

OpenRTM-Aist.

The OpenRTM-Aist [44] framework is a component-based RSF developed at the National Institute of Advanced Industrial Science and Technology (AIST) in Japan. The fundamental component in the OpenRTM framework is the RT-component (Robot Technology). The RT-component defines a common interface for sensor input, component action, state, and output. This RSF somewhat supports architectural primitives since the RT-components which make up the architecture include a built-in FSMs, which is one form of robotic control. OpenRTM rates highly for its use of software engineering principles because of the RT-component specification which adds a high-level of modularity and potential for reuse amongst its components. It is imple-

Table 2. Summary of Robotic Software Frameworks Surveyed.

	Description
OpenRTM	Component based RSF supporting multiple languages and OS, developed by the National Institute of Advanced Industrial Science and Technology in Japan
OROCOS	Consisting of the Kinematics and Dynamics Library, Bayesian Filtering Library and Orocos Toolchain, the OROCOS project is a real-time focused RSF using "modular, run-time configurable software components".
ROS	The Robot Operating System (ROS) is one of the most popular RSFs today currently in its 11th release. ROS is designed to promote collaborative development of robots and enjoys an ecosystem of over 3000 software components for robots written by various contributors.
YARP	Yet Another Robot Platform (YARP) was developed to promote robotic software reuse by providing a rich communications infrastructure for decoupled robotic modules. In contrast with ROS, YARP aims to decentralize control and does not strive to be an operating system for robots.
PX4	The PX4 firmware is "a node-based multithreaded open source robotics framework for deeply embedded platforms". It can be thought of as a real-time capable ROS equivalent, designed to operate as a distributed system with ROS for the control of autonomous unmanned vehicles.

mented in an object-oriented language and makes use of high-level object languages (CORBA). In terms of hardware support, OpenRTM is somewhat supported since it includes roughly 31 different hardware components and a tutorial on creating an RT-component for new hardware. This RSF is compatible with OpenRAVE, which is a 4 year old high-fidelity, 3D robotics simulator. Since it doesn't natively support some of the more modern and advanced simulators, it only earns a "somewhat supported" mark. Logging, playback, and simple debugging are supported through `rtshell`. OpenRTM has limited, built-in fault-tolerance since components can be started and stopped without affecting a running system's stability. While OpenRTM is one of the older and less actively developed RSF's surveyed, its component-based approach is still highly influential today.

Orocos.

Orocos [45] is a general-purpose RSF focused strongly on control and real-time performance in robotics applications. This project consists of three main components: the Kinematics and Dynamics Library (KDL), the Bayesian Filtering Library (BFL), and the Orocos Toolchain. The KDL helps developers model and compute kinematic chains, the BFL provides a set of libraries for the dynamic estimation using Kalman and particle filters, and the Orocos toolchain provides real-time components for robotic applications. Orocos only natively supports FSMs as an autonomy primitive, earning it a value of somewhat supported for architectural primitives. Orocos is particularly well supported in terms of software engineering because it has explicitly stated its design philosophy, uses an object-oriented language, and utilizes a high-level object language (CORBA). It is highly focused on real-time control, and has chosen to outsource other RSF functionalities to other projects. The project has a strong control based theoretical basis in kinematics, dynamics, and Bayesian

estimation. Due to the decision to outsource many RSF functionalities, RSF does not support hardware devices or simulation environments. Furthermore, there is no built-in fault tolerance for OROCOS components, although failure of one component does not mean total failure.

ROS.

ROS [21] represents one of the most complete RSFs available today. ROS is designed to be, as its name suggests, an operating system to simplify the development of robotic software by providing a framework for collaborative development. This framework is built upon a flexible communications middleware which supports message passing between nodes using an asynchronous publish/subscribe design pattern, remote procedure calls (RPC) implemented as services for synchronous messaging, and preemptable RPCs called actions which monitor the progress of an action and report the final result. This communications middleware enables data logging since all messages flowing through the system can be captured. ROS supports multiple architectural primitives such as FSMs, Actions (a ROS specific primitive), Markov processes, and fuzzy decision making. ROS also includes a suite of development tools including several dependency and package management tools, a custom build tool called catkin, and several extensible introspection and visualization tools. Once robotic software is written, developers can test and refine their code using one of the many high-fidelity simulators which work with ROS. Perhaps the most valuable feature of ROS is its library of over 3000 robotic software applications including reusable algorithms and hardware interfaces. This library, known informally as the ROS ecosystem, is a result of a thriving development community which not only contributes packages to the ROS ecosystem but also works on the core components of ROS releasing a major update to ROS yearly. Given the rich set of feature, active

development, and its rate of adoption, many RSF developers have chosen to integrate with ROS to capitalize on ROSs success and bring their platforms to a wider audience.

The RSF receives a somewhat supported value for software engineering because it uses an object-oriented language and can be extended to use high-level object languages like other languages. Its design philosophy is much less clear, however, and does not have a strong theoretical basis in any one area due to its wide scope. While ROS permits dynamic reconfiguration of a node network by shutting down and restarting nodes, the system fails if the master node, which provides naming and discovery services for the network, goes down. There is no explicit fault tolerance in ROS. The communications middleware in ROS is socket based and relies upon a modified versions of the TCP and UDP transport protocols. This, coupled with ROS's handling of memory management, makes real-time operation impossible in its present form.

YARP.

Yet Another Robot Platform (YARP) [18] is an RSF built to prolong the useful life of robotic software and promote interoperability between disparate robotic software components. The YARP development team has authored numerous papers [18, 17, 16] concerning the troublingly low rates of software reuse and high rates of "churn" in the field of robotic software. YARP was designed to address this problem by maximizing interoperability and reuse through its communications middleware and decentralized model. Although YARP does not support many RSF features, YARP is one of the most scalable and flexible RSFs surveyed due to its advanced distribution mechanisms. YARP utilizes a number of carriers which seamlessly and transparently connect applications in a YARP network through socket based technology, shared

memory, and various IPC mechanisms. Since YARP does not attempt to implement many features common to RSF, it appears to score low in many categories. Although this is technically true, the power of YARP lies with its ability to tie together existing software and other RSFs to fill in the gaps it leaves.

PX4.

PX4 [20] is a node-based RSF targeted at unmanned vehicles and can be thought of as a ROS equivalent for deeply embedded, real-time applications. The communications middleware, while much less feature rich, is similar to that of ROS with publish and subscribe message passing facilitated by the μORB object request broker. PX4 is also fully multithreaded which means that, since nodes communicate asynchronously via message passing, applications can be decoupled. This greatly increases the modularity and reuse of control software. PX4 is designed to run on NuttX, a POSIX-like, real-time operating system, meaning that applications can be built to provide real-time performance, a clear advantage over ROS which cannot provide such timing guarantees due to the nature of its communication middleware. PX4 makes good use of software engineering principles since its theoretical background and design philosophy is well documented by its seminal paper. Additionally, it created its own object request broker, makes good use of object-oriented design, and is scoped to permit real-time operation.

Since PX4 is more focused than many RSFs, it lacks many common features such as dynamic configuration of nodes. This is made up for by the fact that PX4 supports a direct interface to ROS running on a companion computer onboard the aircraft. Currently, the two systems can communicate over MAVLink [46], a lightweight communications protocol design for micro-air vehicles (MAVS). In the future, PX4 plans to support the Real-time Publish-Subscribe Wire Protocol [47]. In addition to

the direct interface, PX4 is capable of natively running nodes originally designed for ROS with little modification. The intent of this deep integration is that nodes be tested in ROS, to take advantage of the development and introspection tools of that platform, and then easily transferred to PX4 for its real-time performance once the code is adequately tested. The combination of PX4 and ROS provide a great solution for autonomous UAS and are currently in use on the Dronecode Project, a Linux Foundation collaborative effort to provide a complete UAS platform to include hardware, ground control software, RSFs, and communications protocols. ROS and PX4 can be connected in a distributed system with ROS running computationally complex processes, such as planning and monitoring high-level goals, and PX4 running time-sensitive, safety-critical functions which require real-time performance.

Table 3. Comparison of RSFs Surveyed Using the Kramer Framework.

Category	Criteria	OpenRTM	OROCOS	ROS	YARP	PX4
Specification F1	F1.1 Architectural Primitives	-	-	+		-
	F1.2 Software Engineering	-	-	+	-	-
	F1.2 Architecture Neutrality	✓	✓	✓	✓	✓
Platform Support F2	F2.1 Operating System	J,U,W	U,W	U	U,W	E
	F2.2 Hardware Support	-		+		-
	F2.3 Simulator	-		+	+	+
	F2.4 Configuration Method	RT	RT	RT	RT	Run
Infrastructure F3	F3.1 Low-level Communication	O	O	S,[R]	M,S	O,[R]
	F3.2 Logging Facilities	✓	✓	✓	✓	✓
	F3.3 Debugging Facilities	✓	✓	✓	✓	✓
	F3.4 Distribution Mechanisms	-	-	+	+	-
	F3.5 Scalability	-	-	+	+	
	F3.6 Monitoring Management	-	-	+	+	
	F3.7 Security					
	F3.8 Fault-tolerance	✓			✓	✓
Implementation F1	F4.1 Programming Language	C++,Py,J	C++	C++,Py	C++	C,C++
	F4.2 High-level Language	✓		✓		
	F4.3 Documentation	-	-	+	+	-
	F4.4 Real-time Operation		✓			✓
	F4.5 Graphical Introspection			✓		
	F4.6 Software Integration	✓	✓	✓	✓	✓
	F4.7 Robotic Algorithms	-	+	+		+

2.3.2 Summary.

Table 3 summarizes the analysis of the robotic software frameworks mentioned above. In general, RSF technologies have improved since the initial survey conducted by Kramer in 2007 [25]. Many of RSFs surveyed are interoperable with one another and can leverage each others strengths. ROS is particularly interoperable amongst this set since all of the other RSFs analyzed are, in some way, capable of augmenting or complementing a ROS-based system. The desire to interoperate with ROS is due in large part to its large community of developers, wide adoption, and rich feature set. ROS was founded on the principle of distributed development and has consequently fostered a thriving development community by providing the tools, such as package management, documentation, and promotion necessary for its development. For this reason, ROS constitutes the state-of-the-art RSF and the natural choice for inclusion in a platform with behavioral-flexibility as a design goal.

III. Methodology

This chapter details the implementation of a behaviorally-flexible UAS development platform and an experimental methodology to validate the design in order to answer the research question. The chapter is divided into three sections. The first section addresses IQ2 by describing the design of the platform starting with the major hardware components and then the implementation of software components. The second section describes an experiment, conducted in simulation, which analyzed the effect of arbiter logic and organization on simulated agent performance on navigation-based tasks. This experiment addressed IQ3 since arbiter logic is derived from a corresponding reactive robotic paradigm, allowing inferences about the effectiveness of each paradigm to be drawn from agent performance. The final section concerns the flight testing of the platform. Using a build-up approach, simple agents were flown first to show that this system was safe, stable, and compliant with testing regulations to answer IQ4. Using simulated data from the last experiment, a subset of stable and competent agents were selected for flight testing. Flight tests with these agents were used as the basis of a partial validation of the design. Agents were compared with respect to controller performance as well as qualitative agent behavior in order to answer IQ5.

3.1 Test Item Description

This section lays out the system under test, experimental apparatus, and supporting hardware. Justification is given for the chosen system elements. Figure 7 below depicts the major hardware components of the X8 multirotor used during flight testing of the platform.

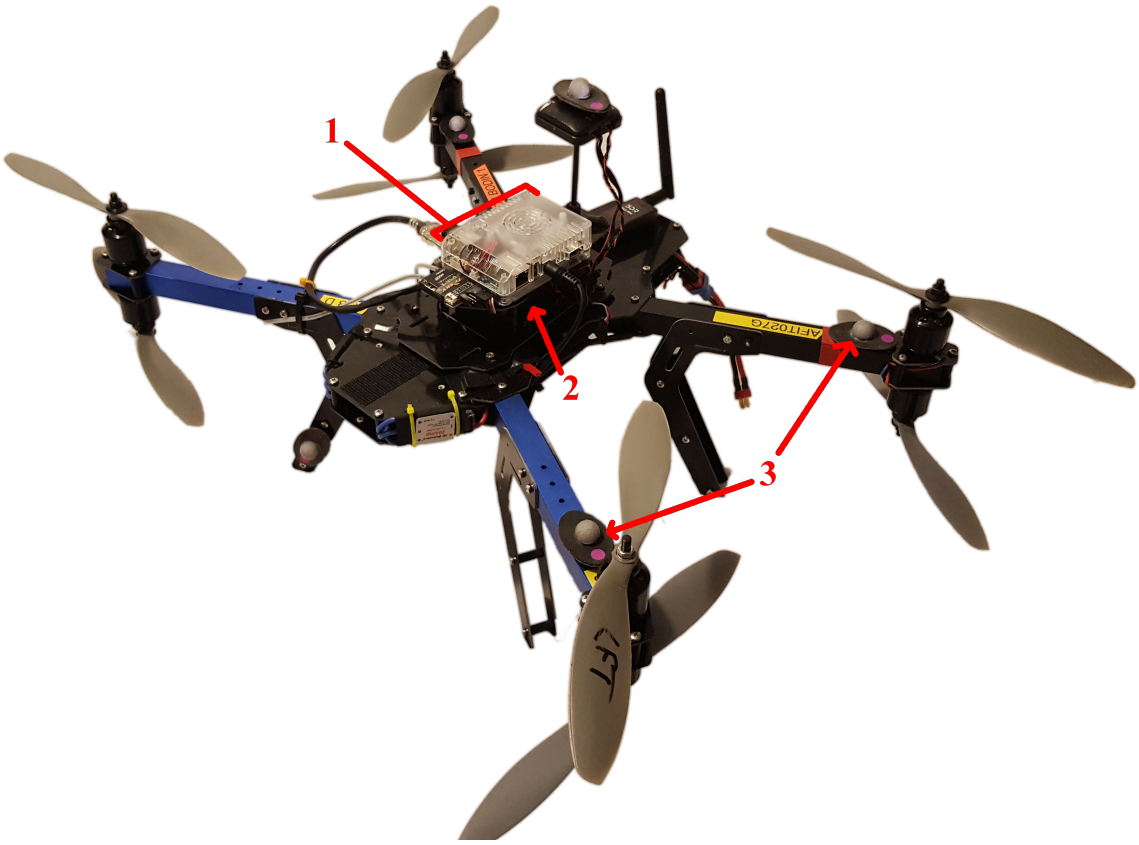

Figure 7. The 3DR X8+ coaxial multirotor UAS. Major components and modifications are annotated in red.

3.1.1 3DR X8+ Coaxial Octorotor.

The 3DR X8+ [48] is a multirotor type UAS with a 4 arm, 600 mm frame carrying two vertically opposed, contra-rotating, 800 kV brushless motors on each arm. The X8+ is is one of the most mature and numerous multirotor platforms in the Autonomy and Navigation Technology (ANT) center inventory. The system was chosen for its maturity, small size, and 15 minute flight time. The following paragraphs, numbered according to Figure 7, highlight the configuration of the vehicle.

1. An Odroid XU4 system on a chip (SOC) computer, similar to Raspberry Pi [49] or BeagleBone Black [50], was secured to the X8+ frame with industrial hook and loop fasteners. The Odroid XU4 was chosen for its small size (83 x 58 x 20 mm), weight (90 g), support of 16.04 LTS Ubuntu Linux (required for ROS), and powerful Cortex-A15 2 GHz and Cortex-A7 Octa core CPU [51]. A majority of the required software was installed using convenience scripts developed by PX4. The specific instructions and files are located at [40]. The same scripts were used to configure the ground control station with the addition of installing QGroundControl, the ground control software developed by PX4.

2. The flight controller used in this research was the Pixhawk [52], one of the most widely used and mature autopilots available. The Pixhawk is powered by a 32-bit Cortex M4 FPU and fail-safe co-processor. The Pixhawk is integrated with high-resolution 3-axis inertial measurement unit (IMU), barometer, GPS receiver, and can support numerous other sensors. The Pixhawk flight controller was chosen for its support of the PX4 firmware, availability, and in-house experience with the flight controller hardware.

3. 5 retroreflective markers were attached to the X8+ frame in an asymmetric pattern. These markers are used in motion capture systems, discussed below,

to reflect infrared light back to cameras for pose estimation. The asymmetric pattern is used to avoid ambiguous orientations, as seen by the motion capture system, which might result in pose estimate errors.

3.1.2 Experimental Apparatus.

The flight testing facility used in this research consisted of a 30' x 30' x 20' indoor Vicon^{TM} motion capture chamber and supporting network infrastructure. The Vicon motion capture system provided sub-millimeter, low-latency position estimates to the PX4 flight controller [53]. Position estimates were first transmitted to the onboard computer via WiFi and then to the PX4 over a serial connection. The Vicon^{TM} estimates were input to the flight controller's Local Position Estimator to estimate the vehicle's position. The indoor Vicon^{TM} motion capture chamber was a natural choice for conducting flight tests due to its large size and precise position estimates.

3.1.3 Ground Control Station.

The ground control station (GCS) for this experimental setup consisted of a laptop running 16.04 LTS Ubuntu. The GCS served three primary purposes. First, the GCS hosted QGroundControl, a flight control and mission planning program used to monitor telemetry and configure vehicle parameters. Per regulation, all actively flying UAS must maintain contact with an associated GCS. During setup, QGroundControl initiated sensor calibration routines and wrote parameter changes to the flight controller. During flight tests, QGroundControl collected and displays aircraft telemetry which was used to monitor vehicle health and performance.

3.1.4 Communications links.

Networking the various computers required two additional devices which are described here. The first was a WiFi connection to the onboard computer from the ground control station to run scripts and to transmit motion capture position estimates as described above. A Netgear Wireless-N 300 Router acted as a wireless access point to extend the existing wired network used to transport Vicon^{TM}, and an Edimax N150 Wi-Fi Nano USB Adapter acted as the wireless network interface for the Odroid XU4.

The second communication link consisted of a MAVLink stream over two different physical networks. The first part of this network consisted of a pair of 3DR 915 MHz wireless modems between the GCS and the Pixhawk. This link served as the primary telemetry downlink to the GCS. The second MAVLink stream existed between the Odroid XU4 and the Pixhawk over a serial connection. This link was used to exchange commands and state information between the Pixhawk and the Odroid XU4 during flight.

3.2 UBF Implementation in ROS

The UBF implementation in ROS grants exceptional flexibility in the configuration of sensors, behavioral logic, and actuators, increasing the adaptability of the platform. This section presents the implemented design using a top-down approach, starting with its high level abstractions and moving toward the individual class implementations. First, a description of the network of ROS nodes comprising the system provides context for the UBF agent node within the larger ROS system. Next, the function of the concrete UBF classes composing an agent node are discussed. Since the structure, function, and theory of the UBF base classes is thoroughly established in [34], this section focuses on modifications and derivations from UBF base classes

specific to this implementation. The section closes with a list of implemented UBF agents used during experimental flight test.

3.2.1 ROS System Description.

The ROS system description represents the highest level of software abstraction for the behavior-flexible development platform. A ROS system is composed of many nodes communicating over topics via messages. While additional overhead is incurred by this form of distributed computing, the node-based, publish and subscribe middleware of ROS offers tremendous behavior-flexibility by way of adaptability. Additionally, the logging and introspection tools utilizing publish and subscribe proved to be an invaluable resource when creating, debugging, and testing this platform. Overhead was minimized where practical by confining the majority of the behavioral logic to a single node. Figure 8 presents the layout of the nodes in the ROS system used in the research. Following is a short description of each node's purpose.

1. ROS master - connects publishing nodes to subscribing nodes [54].

2. ROS parameter server - shared dictionary used by nodes in this system setup to share configuration parameters at runtime. The parameter server can easily be configured using one or more configuration files in the YAML markup language. For example, a configuration file stored parameters in the server which determined hovering altitude, maximum allowed speed, and home location for testing [54].

3. `rosbag` - the primary means of data collection during simulation and flight test. `rosbag` is a logging tool which reads and writes ROS messages to "bag" files [54].

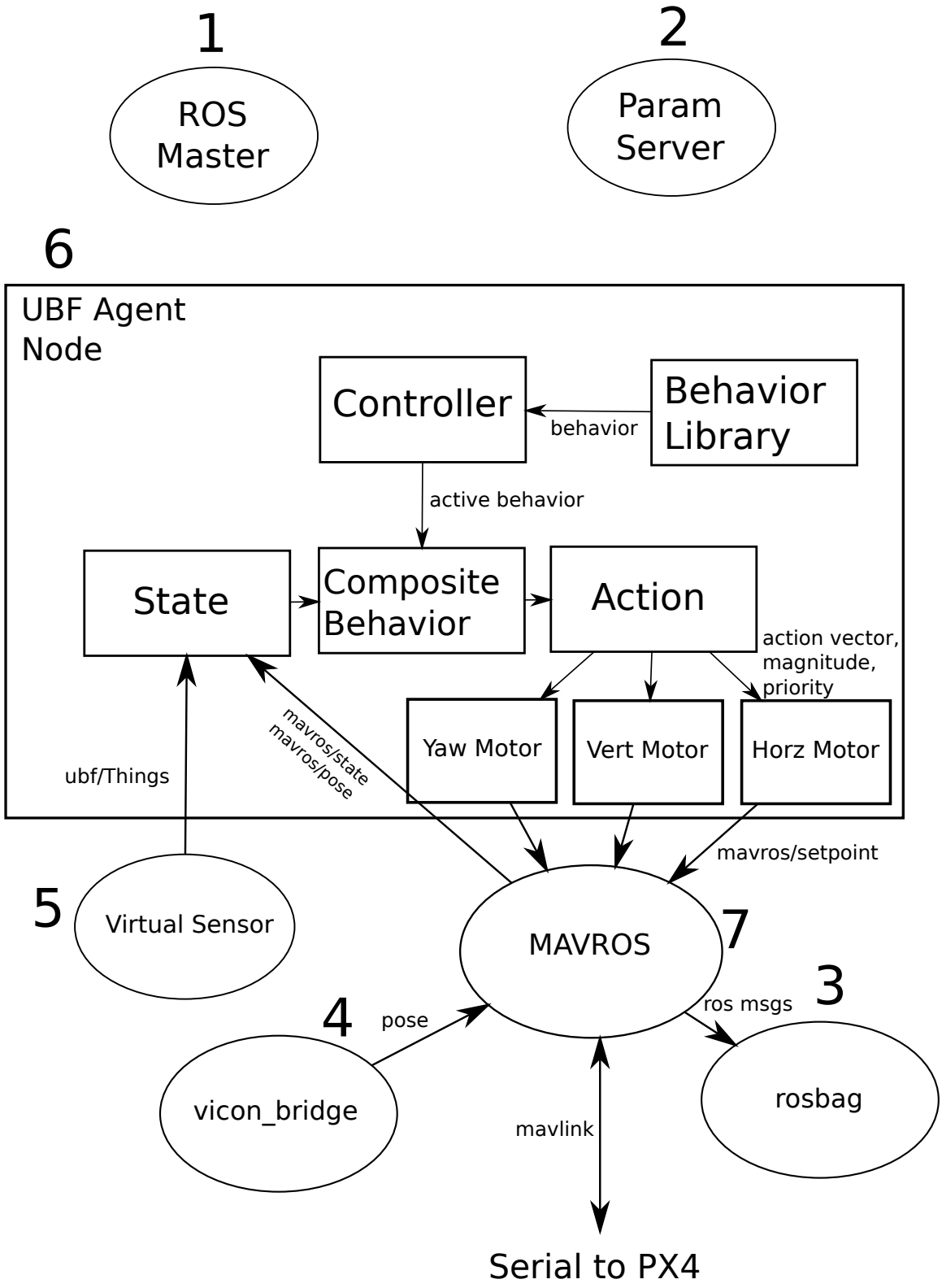

Figure 8. Diagram showing the organization of nodes within the ROS system.

4. `vicon_bridge` - publishes the position and orientation of a subject within a Vicon^{TM}data stream [54].

5. UBF Sensors - virtual sensors used to "sense" the obstacles and targets in the navigation tasks described in this methodology. The UBF sensors package contains four nodes which publish the names and positions of objects according to the task under test.

6. UBF Agent Node - the main node of the system which contains the UBF implementation. The Agent Node serves three main functions. First the UBF node receives and processes sensor messages to build a state representation. Second, the state representation is used by the behavioral and deliberative logic to recommend actions and sequence behaviors. Finally, the winning actions are published to the appropriate topics for execution.

7. `MAVROS` - a communication driver node for the PX4 flight controller which uses the MAVLink protocol. MAVROS acts as an interface between a ROS network and flight controller. MAVROS allows nodes in a ROS network to receive telemetry and system status. A detailed description of the node and examples of its usage can be found at [54].

3.2.2 UBF Agent Node.

The UBF agent node mentioned above was the primary software contribution of this work. This node utilizes a number of classes which allow the UBF agent to receive sensor input, generate actions from UBF behavior logic, and execute actions. The following section describes implementation of UBF classes used in the agents under test and details the next layer of abstraction in the design of the behavior-flexible development platform.

3.2.3 The Controller Class.

The `Controller` class allows the agent to set active behaviors and communicate with other nodes in the ROS system. A `Controller` is able to set the active behavior and generate actions through a `BehaviorLibrary` object which is simply a container for implemented behaviors. Actions generated by behaviors in a `BehaviorLibrary` are converted to an equivalent message and published to `MAVROS`. The `Controller` class can request to arm the flight controller and make mode transition through the `PX4Client`. Although this functionality does not override physical safety mechanisms, such as the arming button on Pixhawk flight controllers, caution should be exercised when attempting to arm and transition modes through a `Controller` object.

3.2.4 The Behavior Class.

The `Behavior` class implements a standard interface to "capture the behavioral logic of the controller as modules..."[34]. This behavior logic is what allows agents to act on sensor data to complete tasks. Behaviors come in two varieties: atomic and composite. Atomic behaviors are the lowest level behaviors possible and produce action recommendations from the current state without relying on other behaviors. Composite behaviors, by contrast, combine one or more atomic behaviors to produce an action recommendation. To arbitrate amongst the conflicting actions recommended by its member atomic behaviors, a composite behavior has an `Arbiter` object, discussed below, to determine a single resultant action based on some logic. Eight concrete behaviors were created for experimental flight test. The intended function of these behaviors is listed below.

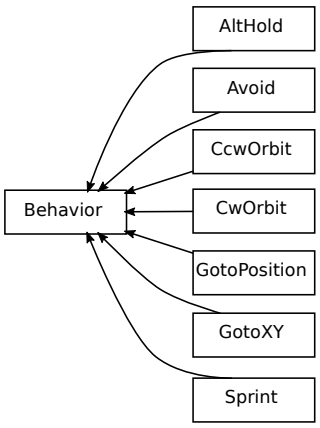

Figure 9. Class hierarchy diagram depicting the implemented atomic behaviors used by agents in this research. Atomic behaviors generate recommended actions based on the current perception of the environment.

Atomic Behaviors.

- `AltHold` - recommends a velocity for the `VertMotor`, which controls vertical motions, to maintain a desired altitude in the local coordinate frame. The priority, given from 0 to 1, of the resulting action is based on how far the current position is from the desired.

- `Avoid` - calculates a repulsive "force" vector in the horizontal plane from each obstacle position known in the current state. A net force vector, formed from the sum of individual force vectors, is used to recommend a velocity to the `HorzMotor`, which controls motion in the XY plane. The priority of this behavior is based on the distance to the closest object.

- `Orbit` - calculates a tangential "force" vector in the horizontal plane from the current target. This force vector specifies the velocity recommended to `HorzMotor`. The `Orbit` behavior is implemented in a counter-clockwise and clockwise variation. Action priority is based on distance to the target.

- `GotoPosition` - sets a position setpoint in the local coordinate frame for the

vehicle. Priority is proportional to the distance from the setpoint beyond a specified threshold and zero otherwise.

- `GotoXY` - sets a velocity setpoint in the XY plane which points toward the current target. Functionally `GotoXY` is very similar to `GotoPosition`, but the logic it uses is quite different. In general, `GotoXY` should be used when speed is the priority and `GotoPosition` should be used when positional accuracy is the priority. Action priority is proportional to setpoint distance above a specified threshold and zero otherwise.

- `Sprint` - like `GotoXY`, `Sprint` sets a velocity setpoint in the XY plane pointing to some position in the local frame. However, `Sprint` does not scale the magnitude of the velocity setpoint as a function of distance to the end point. The intent of this node is to create a behavior that can "sprint" through the finish line of a course to minimize time on course. Priority is proportional to distance to the setpoint.

Composite Behaviors.

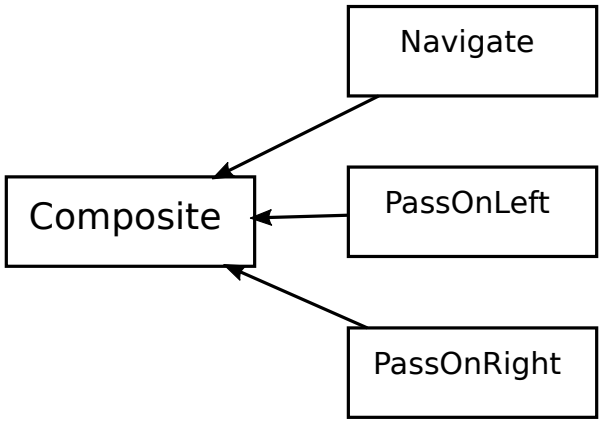

Figure 10. Class hierarchy diagram depicting the implemented composite behaviors used by agents in this research. Composite behaviors are made up of one or more atomic behaviors.

- `Navigate` - This composite behavior combines the `AltHold`, `GotoXY`, and `Avoid` behaviors according to Figure 14. The specific arbiter used to arbitrate actions from `GotoXY` and `Avoid` can be set as desired during instantiation. The resulting composite action produces a velocity based action which attempts to proceed toward a target while avoiding obstacles.

- `PassOnLeft` and `PassOnRight` - These composite behaviors combine the `AltHold`, `GotoXY`, `Avoid`, and `Orbit` behaviors according to Figure 15. The resulting composite action produces a velocity based action which is designed to pass a target on either the right or left and then enter an orbit. When combine with an appropriate deliberator, these composite behaviors are used to run the clover leaf pattern of a barrel race.

3.2.5 The Arbiter Class.

The purpose of the `Arbiter` class is to facilitate action arbitration according to the reactive robotic paradigms discussed in Chapter 2. Arbiters allow multiple atomic behaviors to be combined into a single composite behavior which aides in the modularity and organization of the development platform. Four derived `Arbiter` classes provide these arbitration mechanisms: the `SubsumptionArbiter`, the `PriorityArbiter`, the `PriorityFusionArbiter`, and the `VectorSumArbiter`. Figure 11 below depicts the arbiters implemented for this work. The following paragraphs then describe the arbitration logic of each derived class.

- `SubsumptionArbiter` - this class performs arbitration with respect to the order `Action` objects are added to its action set. For the arbiter to work as intended, behaviors must be added to the composite behavior from the lowest layer to highest. This ordering of behaviors will populate the action set with lower layer actions first. The `SubsumptionArbiter` class allows higher layers to sub-

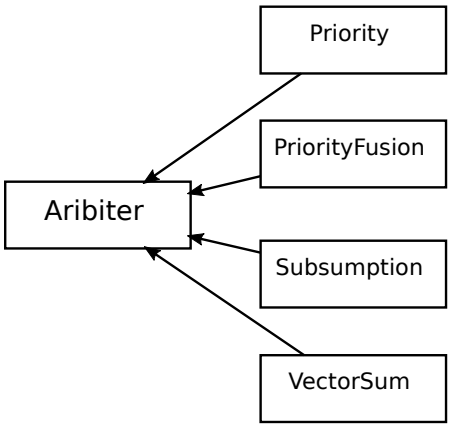

Figure 11. Class hierarchy diagram depicting the implemented arbiters used by agents in this research. Arbiters select or combine a set of actions to form one recommended action according to the reactive robotic paradigm it represents.

sume lower layers by requiring that lower level actions meet a higher activation threshold to vote than higher levels.

- `PriorityArbiter` - this class selects the `Action` object with the highest priority.
- `PriorityFusionArbiter` - this class selects the `Action` object with the highest priority on a per motor basis. For instance, if Action A specifies a horizontal position with priority 0.8 and Action B specifies a yaw angle with priority 0.5, the resulting `Action` object will specify both yaw and horizontal position with the highest component priority which is 0.8.
- `VectorSumArbiter` - this class sums the recommended actions of each `Action` object weight by the priority of that action. For instance, if Action A specifies a horizontal position of (1,0) with priority 0.5, and Action B specifies a horizontal position of (0,1) with priority 1.0, the net horizontal position recommendation would be (0.5,1.0). In the case of a yaw, there is not a obvious way to sensibly fuse yaw recommendation. As such, yaw recommendations are arbitrated on a highest priority basis.

To clarify the function of these arbiters, Figure 12 depicts an example set of actions and the resulting action recommended for each arbiter above.

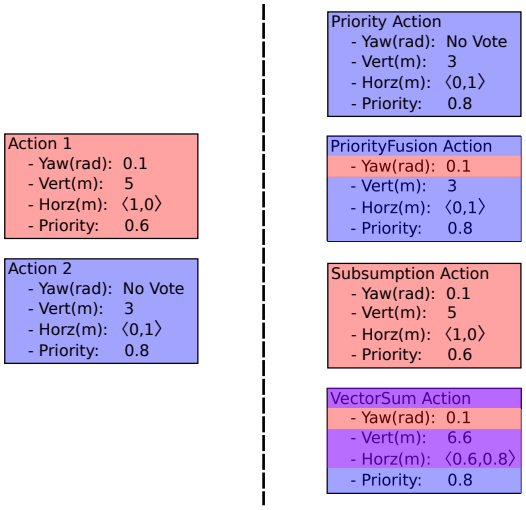

Figure 12. An example arbitration between Actions 1 and 2. The colors correspond to which elements of the original actions (left) are preserved in the final recommend action (right).

3.2.6 The Action Class.

Figure 13 depicts the constituent classes that make up the `action` class.

The `Action` class abstracts the services, movements, or actions offered by a vehicle into a standard interface allowing `Behavior` objects to recommend actions with a confidence level, or activation [34]. `Action` objects are passed up a behavior hierarchy, like the ones shown in Figures 14 and 15, for arbitration by `Arbiter` objects and, eventually, execution by a `Controller` object. To increase modularity, `Action` objects for multirotor aircraft are composed of three `Motor` objects: `YawMotor`, `HorzMotor`, and `VertMotor`, which control motion along a specified vehicle axis according to fields in the `PositionTarget` message [54]. Motion can be specified in terms of position relative to the local coordinate frame or velocity relative to the body frame of the aircraft. When creating an `Action` object, behaviors will set the motors according to

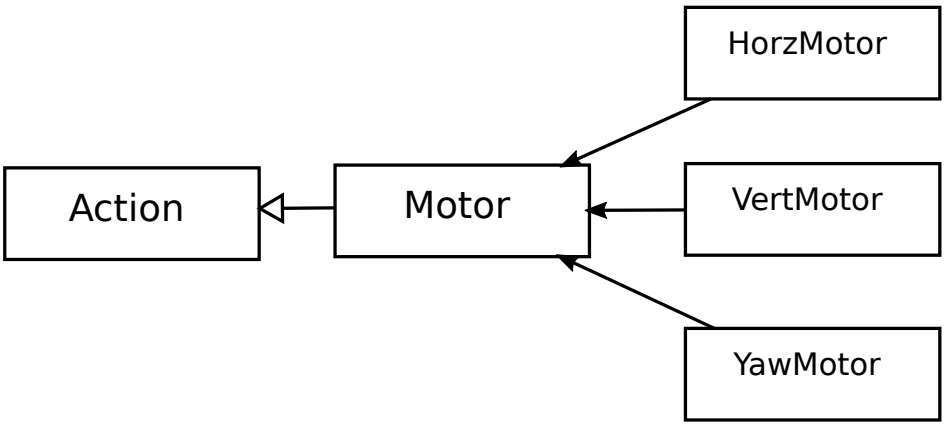

Figure 13. UML class diagram depicting the implementation of the action class. Actions are composed of three motor objects which recommended a movement in a given plane or axis.

the type of motion, i.e. position, velocity, or acceleration, and the magnitude of that motion. Since the `Action` class is modular, extending it to incorporate new actuators is accomplished by simply adding additional derived `Motor` classes. Furthermore, since `Motor` objects encapsulate their own arbitration logic, one can add or subtract motors according to the abilities of the vehicle without having to rewrite the `Arbiter` classes.

3.2.7 The State Class.

The `State` class is responsible for receiving, aggregating, and serving processed sensor data to behaviors and sequencing logic. To illustrate this process, imagine a generic sensor that needs to be integrated into the ROS system for use in a UBF agent. First an appropriate driver ROS node would need to be installed from the ROS ecosystem or created. The driver node would interface with the sensor and publish sensor data on a topic. A ROS Subscriber would be instantiated in the `State` object to receive and process these messages. Once processed, the `State` object would store relevant information to member variables which would then be available, inside

the ROS node, for other objects through public "getter" methods. This research used three different types of sensors to populate the `State` object. The first was the MAVROS node itself, which publishes many topics on the state of the UAS including estimated position, velocity, battery status, and flight controller state. The second was a Gazebo sensor which read model states from a Gazebo simulation to populate target and obstacle information. Finally, virtual sensors, ROS nodes which publish sensor data without interfacing with a real sensor, were used to create a standard environment of obstacles and targets for simulation and flight test.

3.2.8 Implemented Agents.

As stated above, UBF agent nodes contain the behavioral logic and I/O functionality required for the development platform to complete tasks. For this series of tests, five implementations of agent node were created in increasing complexity in order to gain experience with the platform. These agents are listed below.

- `offboard_example` node - an example of offboard control from the PX4 developers guide which commands the multirotor to fly to a position and hold.
- `circle_agent` node - commands the multirotor to fly in a circular pattern. Configuration parameters set the altitude, desired speed, and circle radius for this agent.
- `tf_agent` node - executes a one meter step input followed by a linear frequency sweep in both the horizontal and vertical plane. The combination of these maneuvers was intended to assess controller performance and transfer function characteristics.
- `navigate_agent` node - navigates to a target through an environment filled with obstacles. Figure 14 depicts the behavioral hierarchy for this agent.

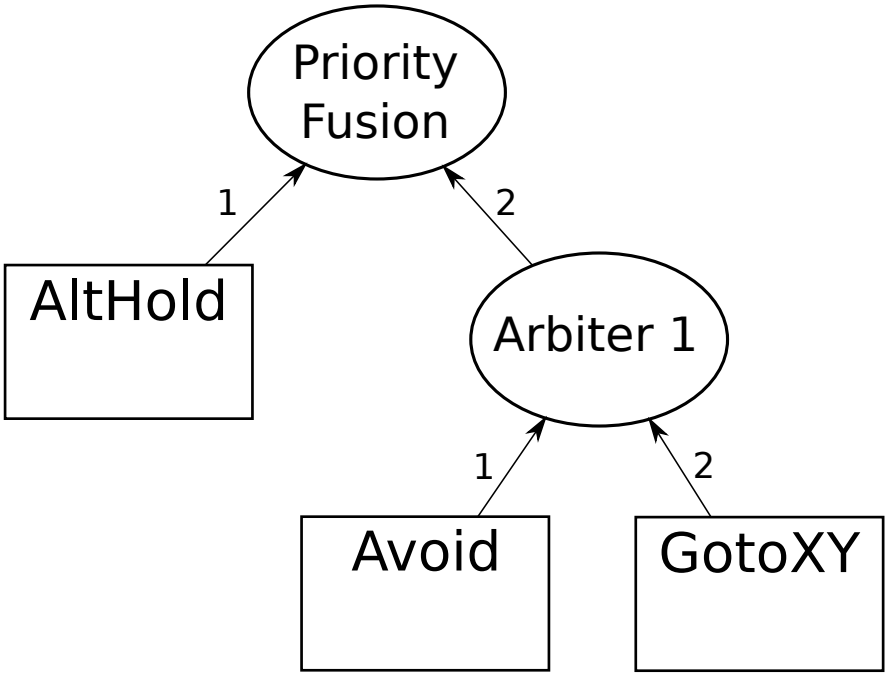

Figure 14. The Navigate behavioral hierarchy. Numbered arrows indicate the order in which behaviors recommend actions to the arbiter which is relevant for the subsumption arbiter. A generic arbiter is given in place of the specific arbiters used to differentiate each agent.

- `barrel_race_agent` node - combines elements of the `navigate_agent` to fly a barrel race course. This is the highest complexity agent since, to perform this task, the agent must sequence behaviors correctly to fly a cloverleaf pattern and arbitrate conflicting behaviors. Figure 15 depicts the behavioral hierarchy for this agent.

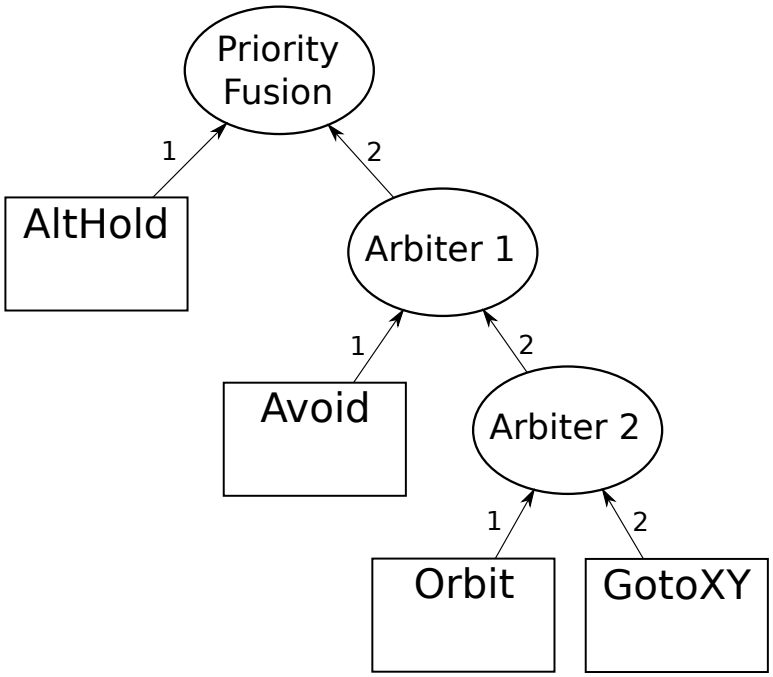

Figure 15. The Pass behavioral hierarchy. Numbered arrows indicate the order in which behaviors recommend actions to the arbiter which is relevant for the subsumption arbiter. A generic arbiter is given in place of the specific arbiters used to differentiate each agent.

3.2.9 Summary.

The platform described above successfully incorporates the UBF into a ROS system capable of controlling a multirotor UAS. The behavioral logic is maintained inside modular agent nodes which can easily be modified to accommodate new sensors, actuators, or behaviors. Although this implementation was designed and tested for multi-rotor aircraft, it could seamlessly transfer to any of the many robotic systems

controlled by PX4 based controllers, such as fixed wing UAS, vertical takeoff and landing craft, or rovers, thanks to the multiple layers of abstraction and modularity inherent architecture.

3.3 Description of Navigation Tasks

A set of courses were designed to assess the robustness and adaptability of the development platform. Three courses were used to test the performance of newly developed agents. The first course consisted of four, one by one meter static obstacles arrange in a box pattern. The agent must proceed through the course from a starting point to a target point 12 meters away and back. The purpose of this course is to assess the baseline performance of UBF agents in a navigation task. Figure 16 below depicts the layout of this course.

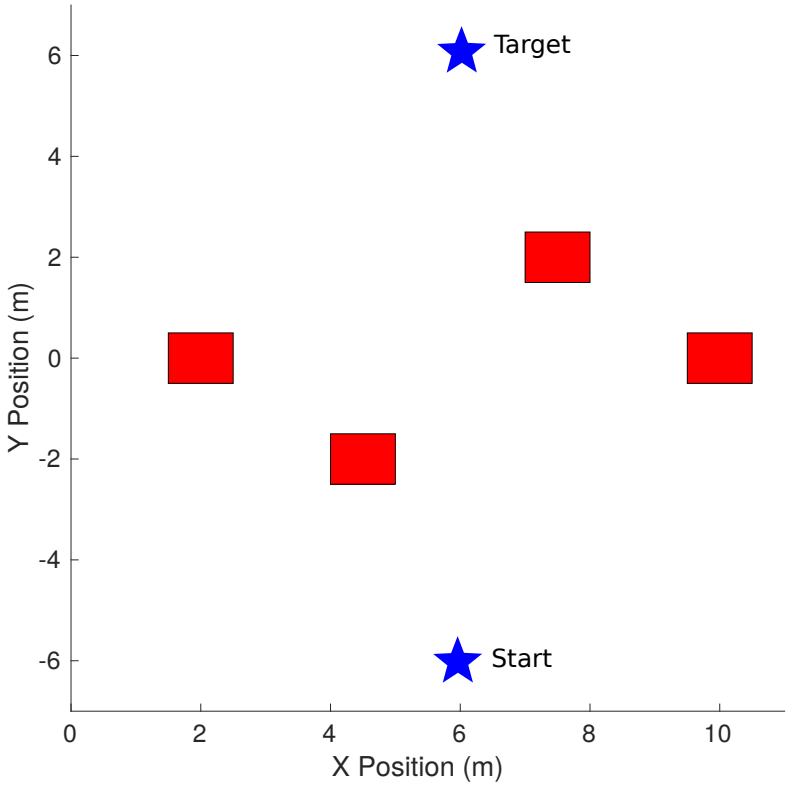

Figure 16. Layout of obstacles in the static obstacle navigation task.

The dynamic navigation course used the same static obstacles as the last course, but introduced an obstacle which moved in a line through the center of the course at one meter per second. The addition of this dynamic obstacle challenges the robustness of agent function. Figure 17 below depicts the layout of this course.

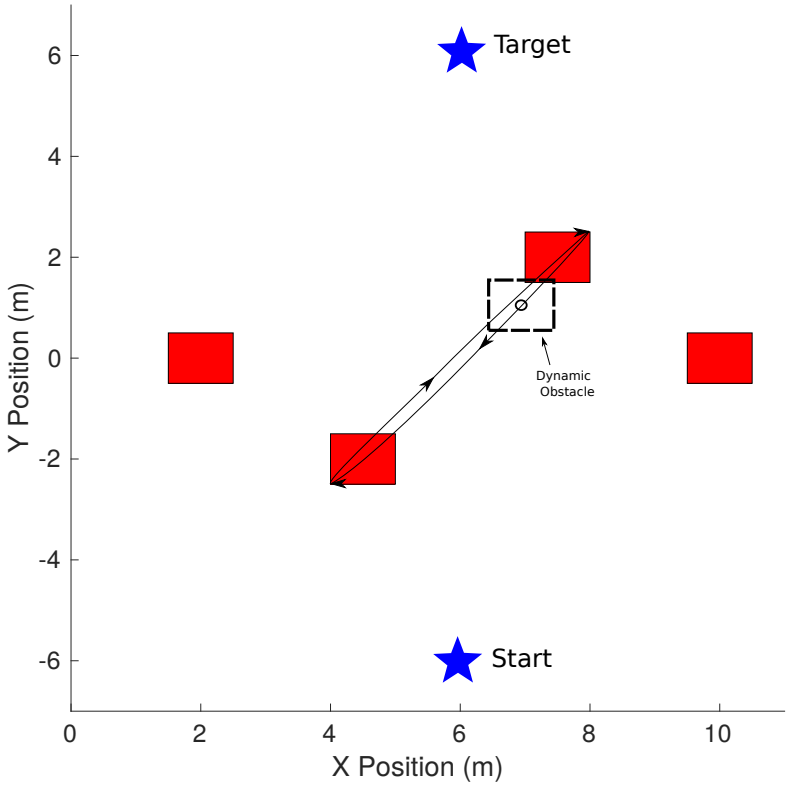

Figure 17. Layout of obstacles in the dynamic obstacle navigation task.

The barrel racing task was based on the popular rodeo event in which competitors ride in a cloverleaf pattern around barrels. Competitors are scored according to total time on course and penalized for hitting barrels. This course was designed to address two research interests. First, this task challenges the platform's adaptability to new tasks since it is appreciably different than the other tasks but relies on most of the same behaviors and it requires the agents to switch the active behavior depending on progress through the course. Second, this task tests the effectiveness of combinations of arbiter logic by forcing agents to arbitrate amongst competing goals. To minimize time on course, it is desirable to cut close to the barrels, minimizing distance, but not so close as to cause a collision. Figure 18 below depicts the course and path for

the barrel race task.

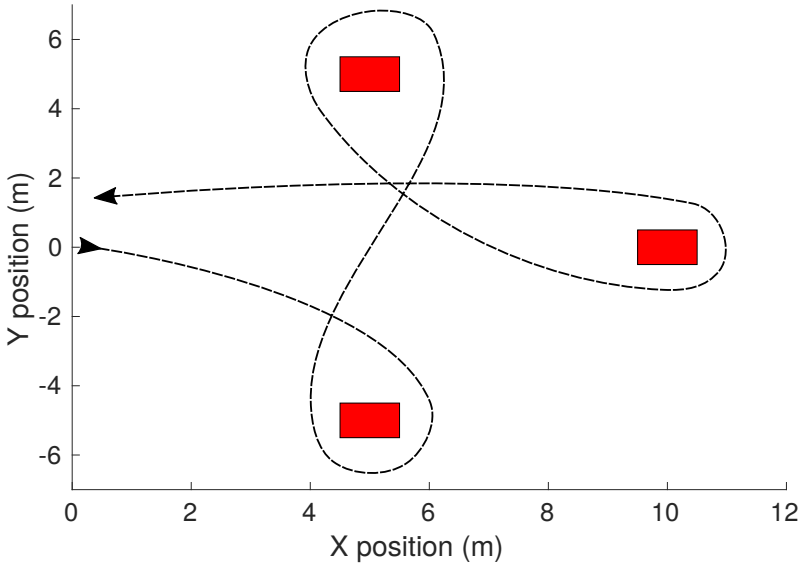

Figure 18. Layout of obstacles in the barrel race course with example agent trajectory.

3.4 Experiment 1: Arbiter Logic and Organization Effect on Simulated Agent Performance

The UBF offers tremendous flexibility in the design and organization of behavior-based agents. Specifically, designers are free to implement and combine the behavior logic, arbitration logic, and hierarchical organization in a way that is appropriate for each task. A drawback of this flexibility is that the design space for creating and tuning agents is massive, even for simple agents. For instance, The architectural elements of Figure 14 and 15 could be rearranged and reimplemented in a countless number of ways while still achieving their intended task. For simplicity, this research sought to explore just one dimension of the design space: the arbiter logic. The purpose of this experiment was two-fold:

1. Answer IQ3 by determining which reactive robotic paradigm, or combination

of paradigms, as expressed by arbiter type, produces the fittest agents for three navigation-based tasks;

2. Answer the research question by validating the UBF implementation in ROS to be a viable, behavior-flexible development platform for simulated multirotor agents.

This section details the experimental design and methodology used to investigate the effect of arbiter variation on simulated agent fitness in the three navigation based tasks. Performance on these tasks served as a basis for comparison for agents of varying arbiter type and arrangement because arbiter logic was the only experimental factor manipulated between the agents tested.

3.4.1 Procedure.

First, the simulated test environment in Gazebo was initialized. The `roslaunch px4 posix_sitl.launch` command [40], launches an emulated PX4 flight controller and Gazebo simulation of a generic quadrotor UAS. The Odroid-XU4 offboard computer, running MAVROS, was connected to the emulated flight controller instance via UDP port 14557. A successful connection was indicated by the terminal output of both the MAVROS process and the PX4 emulation process. Figure 19 depicts the connections of the software-in-the-loop (SITL) environment described above.

The launch file corresponding to the agent under test was then launched. These launch files ran the appropriate simulated sensor node associated with each task, a `rosbag` logging node, the agent node, and loaded configuration parameters. If the `sim` parameter is set to true, the agent will arm and transition to offboard control automatically. The value of the `sim` parameter is safety critical since a real system, if armed, will attempt to takeoff automatically with the agent node running.

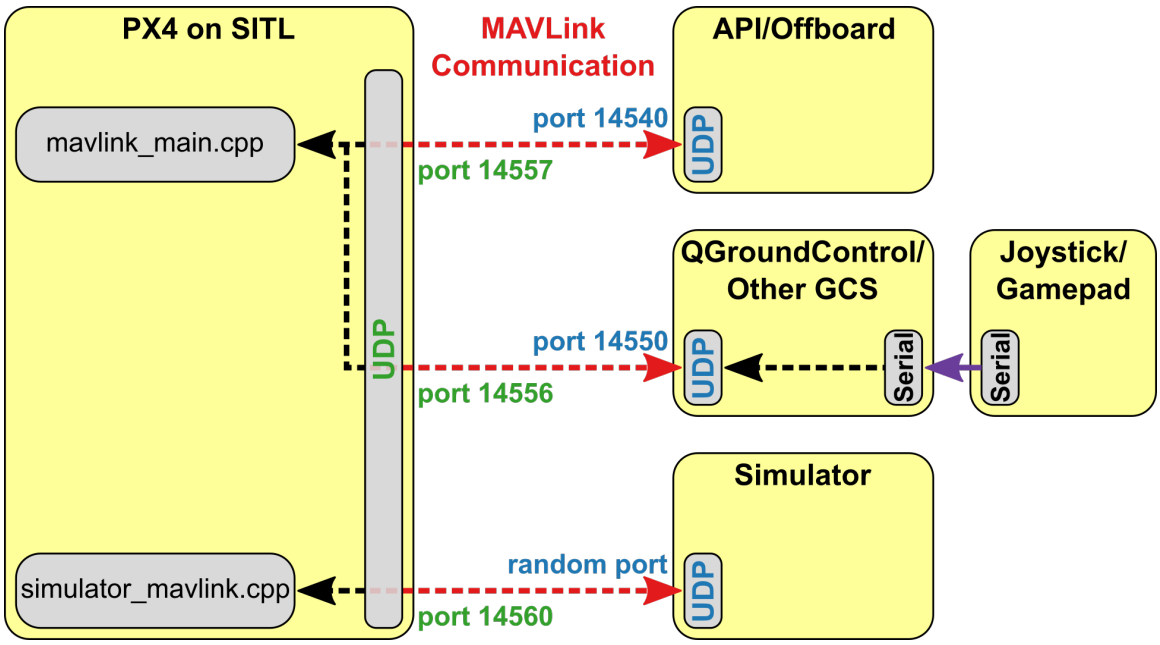

Figure 19. Network diagram showing the connections of components in a "software-in-the-loop" simulation environment [55].

Agent progress was monitored via the Gazebo GUI. A qualitative assessment of agent performance was made with respect to the smoothness of its trajectory, clearance given to obstacles, tendency to get stuck in local minima, etc. The qualitative assessment was primarily used to determine if an agent would be suitable for flight test. After the desired number of runs for the data collection was complete, the agent node process was terminated. Without setpoints being sent, the PX4 firmware returns automatically to the home position and lands.

After verifying the contents of the ROS bag file, this procedure was repeated with a new agent for each agent in the data collection. Table 4 and 5 below depict the test matrix for a replication of navigation and barrel race tests respectively.

3.4.2 Experimental Factors.

In the `navigate_agent` behavioral hierarchy, shown in Figure 14, one arbiter was unspecified. The choice of that arbiter was the experimental factor for the navigation

tasks and could take on any one of the three implemented agents. Likewise, the `barrel_agent` behavioral hierarchy left two arbiters unspecified. The choice of these two arbiters constituted the experimental factors for the barrel race task. Tables 4 and 5 give a complete test matrix of the experimental factors for both tasks in the experiment.

Table 4. Experimental factors for the navigation-based tasks.

Agent #	**Arbiter**
S	Subsumption
P	Priority
VS	Vector Sum

Table 5. Experimental factors for the barrel race task.

Agent #	**Upper Arbiter**	**Lower Arbiter**
S/S	Subsumption	Subsumption
S/P	Subsumption	Priority
S/V	Subsumption	Vector Sum
P/S	Priority	Subsumption
P/P	Priority	Priority
P/VS	Priority	Vector Sum
VS/S	Vector Sum	Subsumption
VS/P	Vector Sum	Priority
VS/VS	Vector Sum	Vector Sum

The agent coding number was determined by the type and position of arbiters used. Note, that the `PriorityFusion` arbiter is functionally the same as a `Priority`

arbiter for both the `navigate_agent` and `barrel_agent` since both child behaviors of the composite are voting on the same motors, and no fusion takes place. The `PriorityFusion` arbiter was removed as an experimental factor for this reason.

3.4.3 Constant Factors.

The factors held constant in this experiment can be divided between factors internal to the agent (UBF factors) and factors external to the agent (environmental factors). Table 9 and 10 present these factors, their anticipated effects, and methods for control below.

Table 6. Factors internal to the UBF agent which are held constant during the experiment. These factors are configuration parameters which could be changed to tune performance without modifying the behavioral logic code itself.

Variable/Factor	Anticipated Effects on Response Variable	How Controlled
Course Geometry	Course geometry will dramatically affect all agent performance metrics	Standardized courses, version controlled code
Behavior Order	Behavior order affects which behaviors will subsume which with a subsumption arbiter	Version controlled code
Repulsive Field	Repulsive field topology and strength affects the "force" pushing each agent which affects the overall emergent behavior and thereby agent performance	Version controlled YAML configuration files
Attractive Field	Attractive field topology and strength affects the "force" pulling each agent which affects the overall emergent behavior and thereby agent performance	Version controlled YAML configuration files
Tangential Field	Tangential field topology and strength affects the "force" tangentially pushing each agent which affects the overall emergent behavior and thereby agent performance	Version controlled YAML configuration files
Behavior Priority Logic	Behaviors determine their own priority somewhat arbitrarily. Priority determines which actions are selected by arbiters which affects overall emergent behavior.	Version controlled code

Table 7. Factors external to the UBF agent which were held constant during the experiment.

Variable/Factor	Desirable Level or Allowable Range	Precision	Anticipated Effects on Response Variable	How Controlled
Vehicle Mass	2.6 kg	1 g	Changes in available excess thrust due to mass change could affect agent performance and controller performance/stability	Standardized Setup Procedure
Vehicle C.G.	N.A.	Not Measured	Extra thrust to counter moments could result in poor performance or instability	Standardized Setup Procedure
Battery Voltage	4.2 v - 3.5 v per cell	10 mV	Insufficient remaining battery energy as indicated by battery voltage could result in degraded thrust or test termination.	Monitoring and changing batteries when low
Initial Clock Offset between GCS and Onboard Computer	<200 ms	tens of milliseconds	The time-stamps of motion capture data, UBF setpoints, and PX4 position estimates may not align. This is known to cause instability in PX4 controllers fusing motion capture data and introduces a confounding variable for response variables relying on time-stamp data.	Network Time Protocol (NTP)
Clock Drift	<200 ms	Not measured	timing accuracy affects the ability of the PX4 flight controller to fuse motion capture estimates which in turn affects the stability of the UAS.	NTP, Assumed to be small
Network Latency between VICON, GCS, and Onboard Computer	<200 ms	1 ms	Network latency will adversely impact PX4 position estimates in the same way as clock offsets.	Assumed to be small, limit traffic to mission critical services, use ethernet when practical
VICON position accuracy	10 cm RMSE	Not measured	Motion capture data will affect controller stability, behavior logic, and data relying on position estimates.	Assumed to be small (∼1 mm)
Atmospheric Conditions	Various	Not Measured	Air density and temperature will affect the excess thrust available to the octorotor which affects agent performance and stability	Indoor test environment

3.4.4 Response Variables.

Table 8 shows the response variables for this experiment.

Table 8. Response variables measured in the arbiter variation experiment (IQ3).

Response Variable	Precision	Relationship to Objective
Course Time	33 ms	Agent performance
Duration of Obstacle Collisions	∼1 mm (spacial) x 33 ms (temporal) resolution	Agent performance

The primary response variables of interest in these tasks were course time and collision duration. These metrics captured the competing interests inherent to the given navigation tasks since the safest route may not be the fastest and vice-versa. Time on task was calculated from the time-stamp of the first action recommendation until the last time-stamp of the last behavior. Actions intended to ready the agent for the next run were not included in this calculation.

Collision duration was calculated as the elapsed time within the boundaries of each obstacle. Thus, behaviors which quickly corrected from a collision were penalized less than those which were slower to respond.

3.4.5 Nuisance Factors.

Relatively few nuisance factors were anticipated for simulated agents. A possible nuisance factor was variability of the state of the simulating computer during a replication of runs. For instance, fluctuations in processor speed or memory utilization on the host computer will degrade fidelity of the simulation environment and could affect the performance of agents in unexpected ways. Multiple replications of the experiment were run to average out variability in simulation quality.

Another potential confounding variable was network latency between the Odroid-XU4 and simulation computer. Latency could significantly impact the performance of agents if timely updates to agent state and action setpoints are not being made.

Since both devices were connected by a wired Ethernet connection, network latency was assumed to be small and fairly constant. Blocking was used to ensure that the network was in a similar state for all agents during a replication.

3.4.6 Known/Suspected Interactions.

The nature of emergent behavior is such that complex interactions between lower level behaviors, priority weights, arbiter choice, and structure are expected and are desirable to produce complex behavior necessary for behavior-based agents to function. These interactions are often unpredictable and, due to the large design space, require metaheuristic optimization techniques to find parameters which produce an effective agent. In this experiment, the number of factors was intentionally reduced to allow for a full factorial experimental design to be conducted. Trends in high-performance agents may elucidate the interactions between arbiter logic and organization in this context.

3.4.7 Assumptions.

The following assumptions were made for this experiment:

1. During a data collection, the state of the simulating computer was relatively constant
2. During a data collection, the state of the network connecting the simulating computer and offboard flight controller was relatively constant
3. The bias and drift of the clocks in the test are sufficiently small to be ignored
4. The starting state of the UAS is consistent between runs

3.5 Experiment 2: Flight Test Validation of the Behavior-Flexible UAS Development Platform

The ability to predict the behavior of physical agents based on their simulated behavior is a crucial feature of UAS development platforms. Systems that rely on emergence are inherently unpredictable. Testing physical agents without being able to vet them in the safety of a simulated environment significantly increases operational risk. Similarly, reactive agents typically require tuning, often by trial and error, before they are proficient at their assigned tasks. If simulated agent behavior accurately reflects physical agent behavior, the need for real-world tuning is reduced which can significantly lower development time, cost, and risk. To assess the degree to which agent performance can be generalized in the UBF/ROS implementation, the following investigation compared and contrasted several metrics of simulated agent performance with that of real-world agents. The objectives of this investigation were as follows:

1. Answer IQ 4 by performing flight tests to validate that the platform is safe, stable, and compliant with testing regulations.

2. Answer IQ 5 by determining the degree to which simulated agent performance generalizes and predicts actual performance

3. Answer IQ 5 by comparing controller performance between simulated and realized systems

4. Answer the research question by validating the UBF implementation in ROS as a viable, behavior-flexible development platform for physical multirotor agents through a flight test demonstration

This section details the experimental design and methodology used to compare the performance of UBF agents and their flight controllers during simulated and phys-

ical task execution. This investigation utilized the three most competent, smooth, and predictable agents identified in the last experiment. To characterize the flight controllers, three additional agents were also flown.

3.5.1 Procedure.

The setup required for flight testing was considerably more involved due to the addition of the multirotor hardware and the external motion capture system. The following paragraphs describe this procedure in detail.

First, the Vicon^{TM}system, GCS, wireless access point, and Odroid were connected to the same local area network. Static IP addresses were assigned to each device which added consistency to the setup procedure. The Vicon system was initialized by powering the camera servers and starting the Tracker software. Next, the Odroid and Pixhawk were powered on and placed in the center of the test chamber.

Next, the ROS system was initialized. A Secure Shell (SSH) connection was initiated to the Odroid to start a ROS Master node via the `roscore` command. The `vicon_bridge` node was then started on the ground control station. Since the ROS Master node was hosted on the Odroid, the `ROS_MASTER_URI` environmental variable on the GCS was set to the IP address of the Odroid.

Third, to ensure that the position messages generated by `vicon_bridge` were being published, a `rostopic echo` command was used to listen to the `~/mocap/tf` topic. `MAVROS` was started over a new SSH terminal on the Odroid with a 962100 baud, serial connection to the FCU. The MAVROS `~/mocap/tf` and `~/local_position/pose` topics were compared while carrying the vehicle around the test chamber to verify that the flight controller was fusing the motion capture estimates properly. Successful fusion was indicated by a match between the position estimates published on these topics to within a few millimeters. If estimates are received but do not match it is

likely that the Odroid clock does not match that of the GCS. The program `rdate` can be used to force the Odroid to synchronize its clock with the GCS.

Finally, individual agents were tested. The safety pilot initiated a manual takeoff to around 2 meters and transitioned the vehicle to altitude hold mode. If the vehicle was stable in altitude hold mode the launch file for the agent under test was executed. Note that these were the same launch files used in the previous experiment with the `sim` parameter set to false to disallow automatic takeoffs. The safety pilot initiated offboard mode, via an assigned RC switch, once the agent began publishing setpoints. Agent progress on each task was video recorded and telemetry data was collected via `rosbag`.

After verifying the contents of the ROS bag file, this procedure was repeated for each agent. The following agents were simulated and flight tested during this experiment.

- `offboard_example`
- `circle_agent`
- `tf_agent`
- Agent VS - Static obstacle course
- Agent VS - Dynamic obstacle course
- Agent VS/VS - Barrel course
- Agent VS/P - Barrel course

After flight testing, simulated tests were carried out to compare to the flight test results. Due to limitations of the flight testing environment two modifications were made to the standard navigation and barrel race tasks. First, course sizes were

reduced to allow for adequate clearance from the flight test chamber walls. Second, maximum agent speed was reduced from 5.0 m/s to 0.5 m/s to allow the safety pilot time to recover the UAS in the event of anomalous behavior. The new simulations reflected these modifications to provide an unbiased comparison.

3.5.2 Experimental Factors.

For this experiment, seven agents were tested in simulation and during flight test. Per the build-up approach, the simplest agents were flown first. The first agent was the `offboard_example` agent which commanded the multirotor to go to a specific position two meters above the chamber floor and hover. The second agent was the `circle_agent` which commanded the multirotor to fly in a large circle at 1 m/s to test the path tracking ability of the flight controller. The third agent was the `tf_agent` which commanded step functions and frequency sweeps to characterize the flight controller. The fourth agent was the `navigate_agent` with a vector summation arbiter. This agent ran the static and dynamic navigation courses. The sixth agent was `barrel_agent` with vector summation arbiters which ran the barrel race course. Finally, the seventh agent was the `barrel_agent` with subsumption and priority arbiters which also ran the barrel race course. These agents were selected as good candidates for flight test based on the smoothness of their trajectories and their fitness determined in the previous experiment.

3.5.3 Constant Factors.

The factors held constant in this experiment can be divided between factors internal to the agent (UBF factors) and factors external to the agent (environmental factors). Table 9 and 10 present these factors, their anticipated effects, and methods for control below.

Table 9. Factors internal to the UBF agent which are held constant during the experiment. These factors are configuration parameters which could be changed to tune performance without modifying the behavioral logic code itself.

Variable/Factor	Anticipated Effects on Response Variable	How Controlled
Course Geometry	Course geometry will dramatically affect all agent performance metrics	Standardized courses, version controlled code
Behavior Order	Behavior order affects which behaviors will subsume which with a subsumption arbiter	Version controlled code
Repulsive Field	Repulsive field topology and strength affects the "force" pushing each agent which affects the overall emergent behavior and thereby agent performance	Version controlled YAML configuration files
Attractive Field	Attractive field topology and strength affects the "force" pulling each agent which affects the overall emergent behavior and thereby agent performance	Version controlled YAML configuration files
Tangential Field	Tangential field topology and strength affects the "force" tangentially pushing each agent which affects the overall emergent behavior and thereby agent performance	Version controlled YAML configuration files
Behavior Priority Logic	Behaviors determine their own priority somewhat arbitrarily. Priority determines which actions are selected by arbiters which affects overall emergent behavior.	Version controlled code

Table 10. Factors external to the UBF agent which were held constant during the experiment.

Variable/Factor	Desirable Level or Allowable Range	Precision	Anticipated Effects on Response Variable	How Controlled
Vehicle Mass	2.6 kg	1 g	Changes in available excess thrust due to mass change could affect agent performance and controller performance/stability	Standardized Setup Procedure
Vehicle C.G.	N.A.	Not Measured	Extra thrust to counter moments could result in poor performance or instability	Standardized Setup Procedure
Battery Voltage	4.2 v - 3.5 v per cell	10 mV	Insufficient remaining battery energy as indicated by battery voltage could result in degraded thrust or test termination.	Monitoring and changing batteries when low
Initial Clock Offset between GCS and Onboard Computer	<200 ms	tens of milliseconds	The time-stamps of motion capture data, UBF setpoints, and PX4 position estimates may not align. This is known to cause instability in PX4 controllers fusing motion capture data and introduces a confounding variable for response variables relying on time-stamp data.	Network Time Protocol (NTP)
Clock Drift	<200 ms	Not measured	timing accuracy affects the ability of the PX4 flight controller to fuse motion capture estimates which in turn affects the stability of the UAS.	NTP, Assumed to be small
Network Latency between VICON, GCS, and Onboard Computer	<200 ms	1 ms	Network latency will adversely impact PX4 position estimates in the same way as clock offsets.	Assumed to be small, limit traffic to mission critical services, use ethernet when practical
VICON position accuracy	10 cm RMSE	Not measured	Motion capture data will affect controller stability, behavior logic, and data relying on position estimates.	Assumed to be small (∼1 mm)
Atmospheric Conditions	Various	Not Measured	Air density and temperature will affect the excess thrust available to the octorotor which affects agent performance and stability	Indoor test environment

3.5.4 Response Variables.

Regulatory compliance was determined by the system's ability to meet four safety critical requirements. The first requirement is that the safety pilot be able to safely transfer control to the UBF agent when desired and transfer control from the UBF agent at any time. Second, the system must be able to achieve, maintain, and safely terminate flight. Finally, the system must maintain connectivity with a ground control station. Table 11 summarizes these requirements.

Table 11. Requirements used to determine system safety, stability, and compliance with testing regulations (IQ4).

Requirement	Pass/Fail
Transfer of Control	
Achieve Flight	
Maintain Flight	
End Flight	
GCS Connection	

Agent and controller performance was assessed according to a diverse set of metrics. Table 12 shows the response variables for agent comparisons.

Table 12. Comparison metrics used to determine the similarity of the simulated and real agent behavior (IQ5).

Comparison Metrics	Precision	Relationship to Objective
Rise Time	33 ms	Compare responsiveness (IQ5)
Latency	33 ms	Compare responsiveness (IQ5)
Cutoff Frequency	1 Hz	Compare controller performance (IQ5)
Obstacle Collision Duration	1 mm (spacial) x 33 ms (temporal) resolution	Compare agent performance (IQ5)
Agent Course Time	33 ms	Compare agent performance (IQ5)

These metrics were selected in an attempt to account for observed differences in agent behavior and performance between simulation and physical flight testing. No direct, quantitative comparison between the simulated and real agent trajectories was attempted for this experiment. [56] presents method for quantitative comparison

of robot trajectories, but this technique was not utilized due to its complexity and time limitations. Mean course time and mean collision duration are used instead to quantitatively compare agent behavior. Additionally, since perception and behavioral logic was held constant between runs, variance due to the system itself was likely the result of different hardware and flight controller tuning. An analysis of the platform as a linear system was intended to quantify these differences.

3.5.5 Nuisance Factors.

Field testing with hardware introduces a wide variety of confounding variables when compared to simulation alone. First and foremost, the simulation operates on an idealized model of multirotor dynamics as discussed in Chapter 2. Certain aspects of the real-world dynamics are not present in simulation and will affect the system response as measured by the transfer function metrics given above. Another possible nuisance factor was the initial configuration of the multirotor. Battery voltage, changes to the center of gravity, and initial location all affected the performance of the SUAS. To control for these effects, batteries were changed frequently between runs and located by hook and loop fastener to provide a consistent center of gravity. Many environmental factors were minimized since testing took place indoors. There should be little variability in the location of the target, static, and dynamic threats since they were purely virtual precepts to the agent and positioning information was very precise.

3.5.6 Assumptions.

The following assumptions are made for this experiment:

1. Since the vehicles operated within a benign region of their flight envelopes, i.e. low speed and gentle maneuvers, differences in aircraft performance capabilities

did not contribute to observed performance differences.

2. Controller tuning was not a large contribution of observed differences since standard tuning procedures were employed and kept constant across runs.

3. The performance of the emulated autopilot was sufficiently analogous to the performance of a real autopilot such that no agent configuration was favored over another.

4. VICON positioning data were sufficiently accurate for the agents to navigate the scenario correctly.

5. All hardware and firmware function correctly during the test.

3.5.7 Limitations.

As is often the case with flight testing, there were several limiting factors which constrained the scope and depth of this experimental design. Safety and test chamber size were some of the largest limiting factors. At the request of the safety pilot, the maximum operating speed was lowered from 5.0 m/s, as was used in the first experiment, to 0.5 m/s. Additionally, the geometry of tasks had to be compressed to provide an adequate buffer from the chamber walls. These limitations were largely overcome by modifying the simulated tasks to reflect these modifications. Safety also dictated that certain agents would not be suitable for flight test. Priority and subsumption based agents in particular were disallowed due to their "jerky" flight patterns.

Time also constrained the experiment. A full factorial experimental design required only three agents per repetition for the navigation tasks and nine agents per repetition for the barrel race task. These 15 agents were each allowed five runs per

data collection which necessitated 75 runs per replication of the experiment. Although the 75 runs is manageable for simulation, flight testing of all agents was deemed infeasible due to battery capacity limitations, operational safety, availability of test personnel, and chamber size. This limitation was overcome by selecting the safest and highest performing agents for flight test and limiting the number of runs for each.

3.5.8 Summary.

The results of a comparison of simulated and physical agents offer significant insight into the practicality of the platform as a whole. At all stages of development it is important to test agent performance for safety, competence, and correct function in simulation before risking hardware and time in flight tests. As mentioned above, predictability of physical agents relies heavily to some degree how well simulated performance translates into actual performance. Additionally, these result might offer insights into the generalizability of the UBF to other physical systems since the vehicle flown in the experiment, an octorotor, is a relatively different from the simulated hardware, a quadrotor. Finally, and importantly, this experiment demonstrates that platforms well for physical agents.

IV. Results

The following chapter discusses the results and significant findings of this research. Two experiments were conducted in order to answer the research question, "Is it possible to develop a behavior-flexible development platform for autonomous UAS agents using open-source software components?" The first experiment assessed the performance of a set of UBF agents on three navigation-based tasks in a simulated environment. This experiment was designed to identify which reactive robotic paradigms, as represented by the different arbiter types, performed best for navigation tasks. The second experiment consisted of a flight test program of seven UBF agents on various tasks. This experiment sought to establish the regulatory compliance of the platform and compare simulated results with flight test results to gauge the predictability of the platform.

For this analysis run times were assumed to be approximately normally distributed. This assumption was validated with a chi-squared goodness-of-fit test which determined that a quarter of run times were normally distributed to a confidence of 95% and half of all run times were normally distributed to a confidence of 90%. Runs which did not appear to be normal were typically bimodal, with each cluster being approximately normal.

The remainder of the chapter is presented in two parts. First, the arbiter logic and organization effect on simulated agent performance is presented. The results of each individual task are given first, followed by a discussion of significant findings, trends, and other analysis. Second, the flight test validation of the platform is recounted starting with the simplest agents and building to the most complex. This section also concludes with a discussion of significant findings, trends, and analysis.

4.1 Experiment 1: Arbiter Logic and Organization Effect on Simulated Agent Performance

The following sections present the results of the first experiment of this research. This experiment assessed the performance of simulated UBF agents on three navigation based tasks to determine which combination of arbiter logic and organization completed tasks in the least time and shortest total duration of obstacle collisions. Since an arbiter's logic is derived from its corresponding reactive robotic paradigm, this experiment offers insight into the effectiveness of each paradigm for UAS agents on navigation tasks.

4.1.1 Static obstacle navigation results.

The static obstacle navigation task consisted of navigating from a starting point to a target point and back while avoiding static obstacles. Figure 20 depicts the mean obstacle collision duration per run for each agent. Collision duration was measured by the number of position estimates (captured every 33 ms) within one by one meter square around each obstacle.

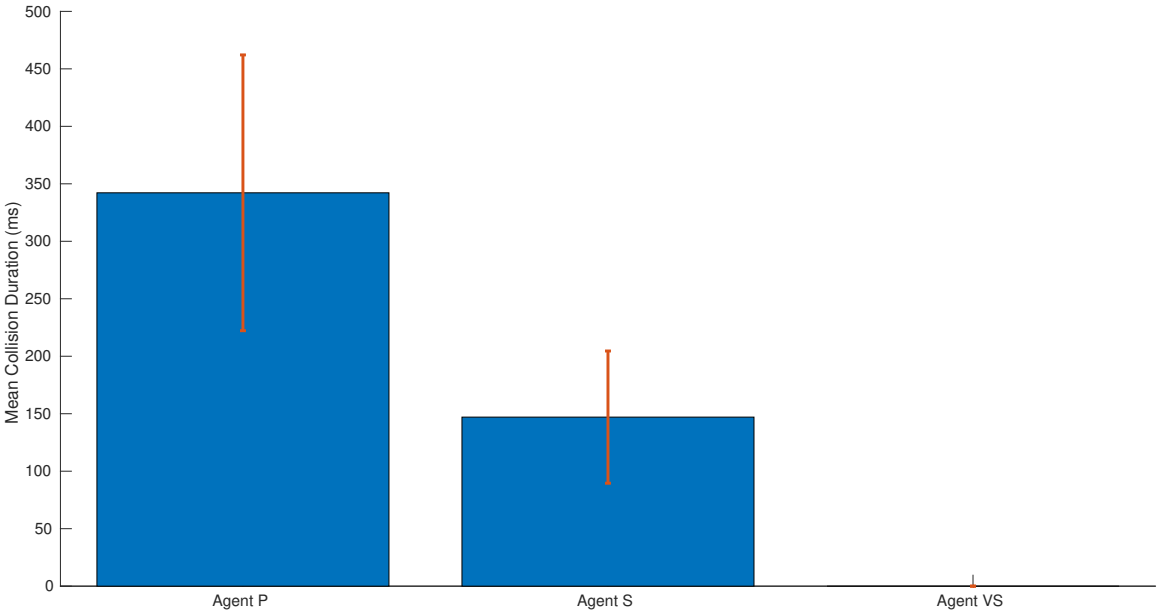

Figure 20. Bar chart depicting mean collision duration per run of the static obstacle navigation task for each agent.

Agent time on task was calculated by taking the difference of the starting setpoint, as indicated by a flight controller mode change, to the final setpoint, also indicated by a mode change. The mode transition for the starting setpoint was triggered by being within a radial distance of 0.5 meters of the start location with a maximum speed of 0.2 m/s for a consistent starting state between runs. The final setpoint mode transition was triggered when the UAS was within 0.5 meter of the start location with any speed. Figure 21 presents the average agent time on task per run.

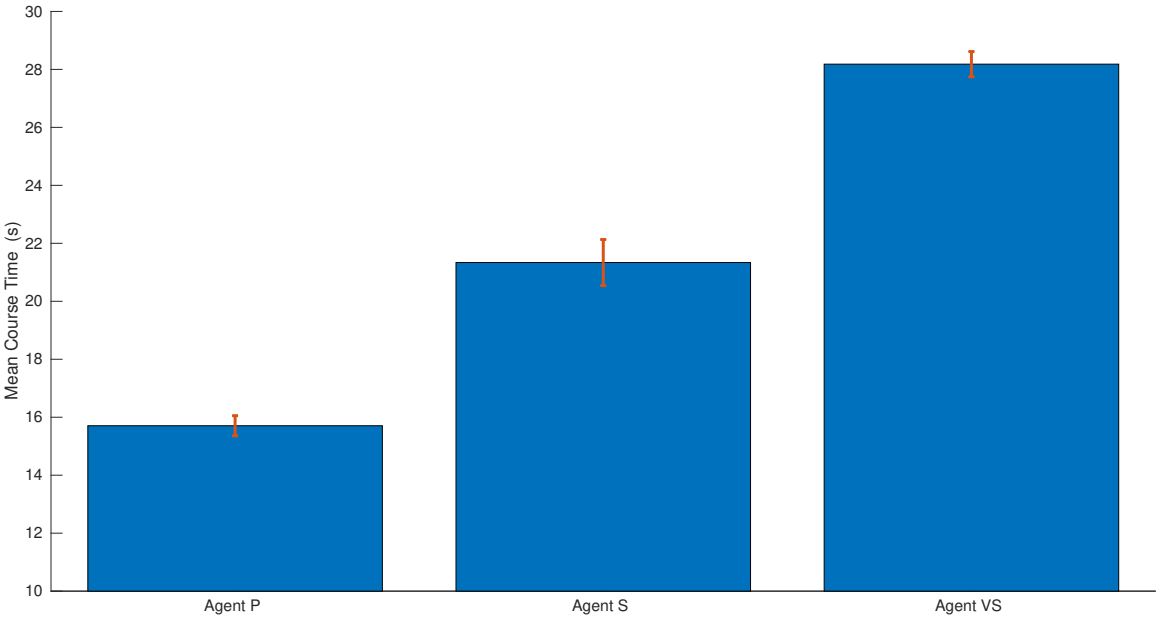

Figure 21. A bar chart depicting mean time to complete the static obstacle navigation task for each agent.

Tables 13 and 14 present a statistical analysis of agent mean time on task and collision duration respectively for the static obstacle navigation task.

Table 13. Statistical summary of mean completion time on the static obstacle task.

Agent	Trials	Mean (s)	CI 95%	Std. Dev.
S	114	21.34	[20.54, 22.13]	4.28
P	122	15.71	[15.36, 16.05]	1.94
VS	131	28.18	[27.75, 28.61]	2.52

Table 14. Statistical summary of mean collision duration on the static obstacle task.

Agent	Trials	Mean (ms)	CI 95%	Std. Dev.
S	114	147	[89, 204]	310
P	111	342	[22, 462]	669
VS	118	0	[0, 0]	0

These results indicate that Agent P (with a priority-based arbiter) is the fastest and riskiest agent. Agent VS (with a vector summation arbiter) is the slowest and safest. This result is directly attributable to arbiter logic. The Agent P took a more direct route between targets since its `GotoXY` behavior was "winning" or "active" since it was allowed to publish the action which was executed most of the time. `Avoid` only overruled (i.e. had a higher priority than) `GotoXY` at the last possible instance to avoid collisions. Agent P exhibited jerky, erratic, and error-prone behavior since actions changed rapidly. By contrast, Agent VS incorporated recommendations from `Avoid` at all times through its summation mechanism. The resulting behaviors were much more smooth and gave obstacles ample clearance. The smoothness of action made this agent a logical choice for further flight testing since the agent is less likely to make an erratic, unsafe movement. This conclusion does not rule out Agent 2 as useful, however. Depending on the application, the need for speed, and severity of a "collision" state, the priority agent could be preferable over Agent VS.

4.1.2 Dynamic navigation results.

The dynamic obstacle navigation task was the same as the static version with the addition of a dynamic obstacle which traveled back and forth along a line. The experiment consisted of six data collections with approximately 100 total runs for each agent. Figure 22 depicts the mean collisions duration and Figure 23 present the

mean time on task for each agent.

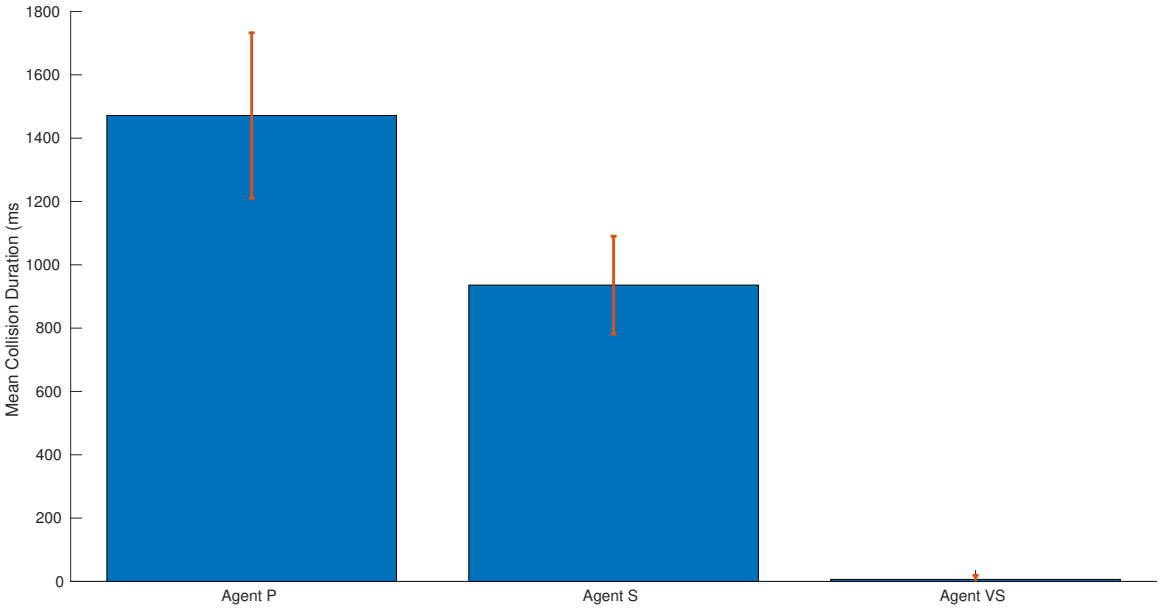

Figure 22. Bar chart depicting mean collision duration per run of the dynamic obstacle navigation task for each agent.

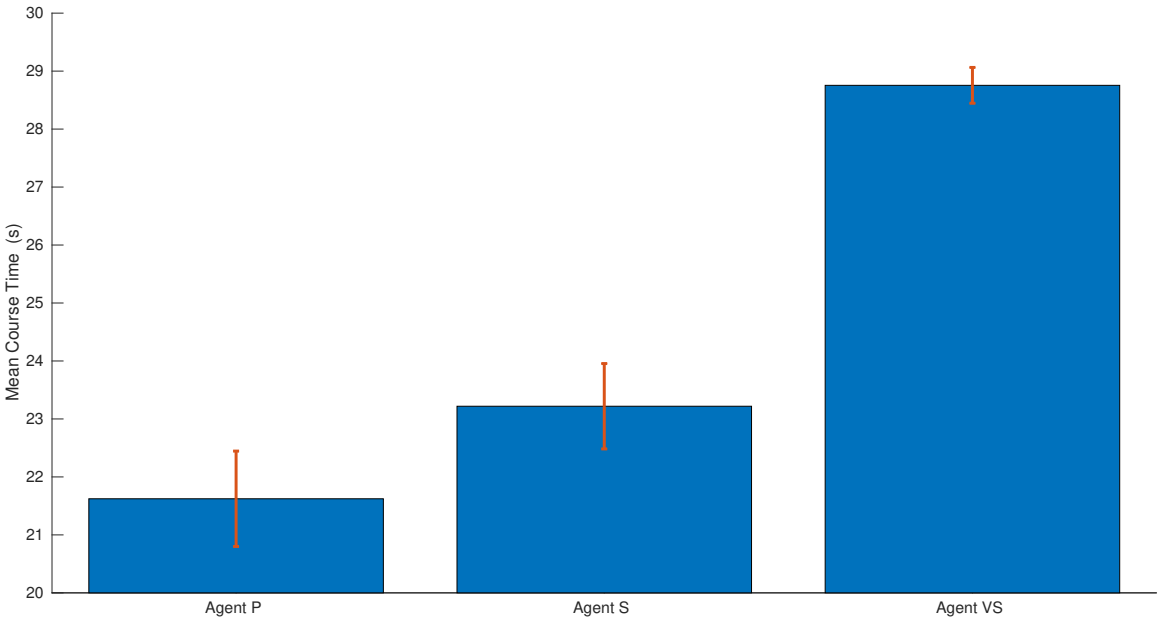

Figure 23. A bar chart depicting mean time to complete the dynamic obstacle navigation task for each agent.

Tables 15 and 16 present a statistical analysis of agent mean time on task and collision duration respectively for the dynamic obstacle navigation task.

Table 15. Statistical summary of mean completion time on the dynamic obstacle task.

Agent	Trials	Mean (s)	CI 95%	Std. Dev.
S	125	23.22	[22.48, 23.96]	4.00
P	111	21.62	[20.81, 22.45]	2.44
VS	118	28.75	[28.45, 29.06]	5.52

Table 16. Statistical summary of mean collision duration on the dynamic obstacle task.

Agent	Trials	Mean (ms)	CI 95%	Std. Dev.
S	125	936	[781, 1090]	872
P	111	1472	[1211, 1733]	1388
VS	118	6	[0, 19.17]	70

Again, Agent VS was the safest and Agent P was the fastest. All agents were slowed by the additional obstacle and incurred more collisions, but not dramatically more. Most collisions were on the new dynamic obstacle. This could be due to the fact that only the position and not the trajectory of obstacle was considered for avoidance maneuvers. More sophisticated behavioral logic which predicted the obstacle's future position and maneuvered accordingly could address this shortcoming.

The dynamic navigation task was intended to assess the robustness and behavioral-flexibility of the designed UBF agents. No modifications were made to agent logic despite the addition of a new type of dynamic obstacle. The ability to cope with dynamic, unpredictable environments is a key requirement for autonomous UAS. Any difference in the mean collision duration per run or mean time to complete a course would indicate how robust an agent was to the modification of this task. Figures

24 and 25 show the performance differences of each agent between the static and dynamic tasks.

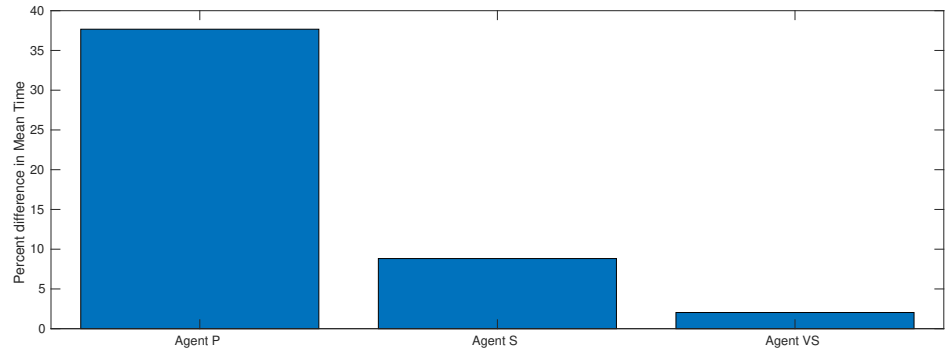

Figure 24. Bar chart comparing percent difference in mean agent time between the static and dynamic obstacle navigation tasks. Lower is better.

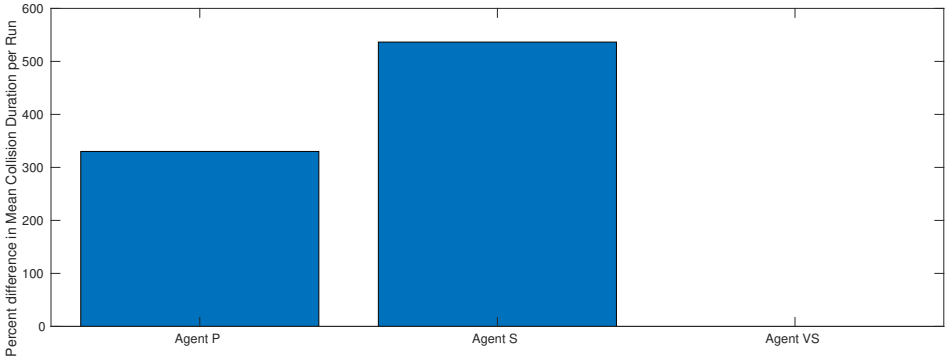

Figure 25. Bar chart comparing percent difference in mean collision duration per run between the static and dynamic obstacle navigation task. Lower is better.

These figures indicate that Agent VS was the most robust to the additional, dynamic obstacle since runs were only approximately 3% longer and collision duration increased only slightly. The Agent S, the subsumption agent, was the least robust as mean run-time increased by over 10% and the percent increase in collision duration was nearly 14%.

4.1.3 Barrel Race results.

The barrel race task involved navigating a three-point course along a set path while avoiding collisions with simulated barrels at each point. This task required the addition of the `orbit` behavior to allow agents to round the barrels, which permitted the use of an additional arbiter in the behavioral hierarchy. This additional arbiter choice increased the test space from three to nine agents composed of permutations of arbiter type. Since velocity limits were imposed on agent movements, variability in time on task was largely due to differences in the agent's path. Agents that flew close to the barrels saved distance, and therefore time, but also were at risk of collisions. Despite these competing interests, agents rarely collided with the barrels during the 100 runs comprising the experiment. This result is likely due to the sparse distribution of obstacles on the course. Individual agents could have been tuned to reduce the barrel turn radius, but a fixed set of parameters were chosen so that each agent could be compared on an objective basis. The performance of the nine agents tested is shown in Figures 26 and 27 below.

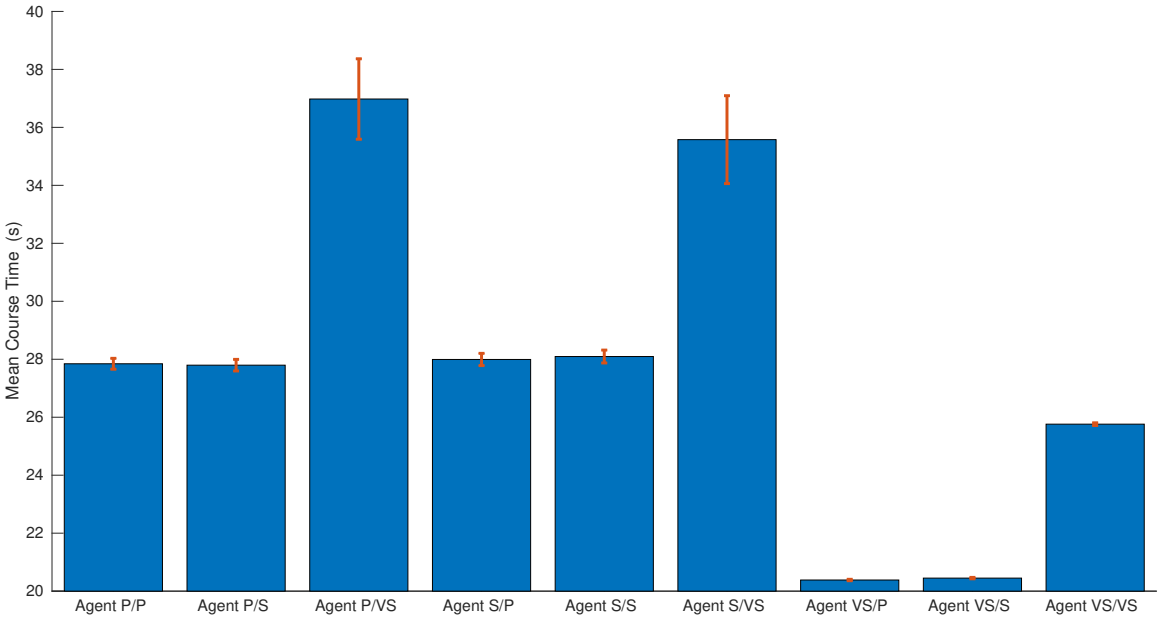

Figure 26. Bar chart depicting mean time to complete the barrel race task for each agent.

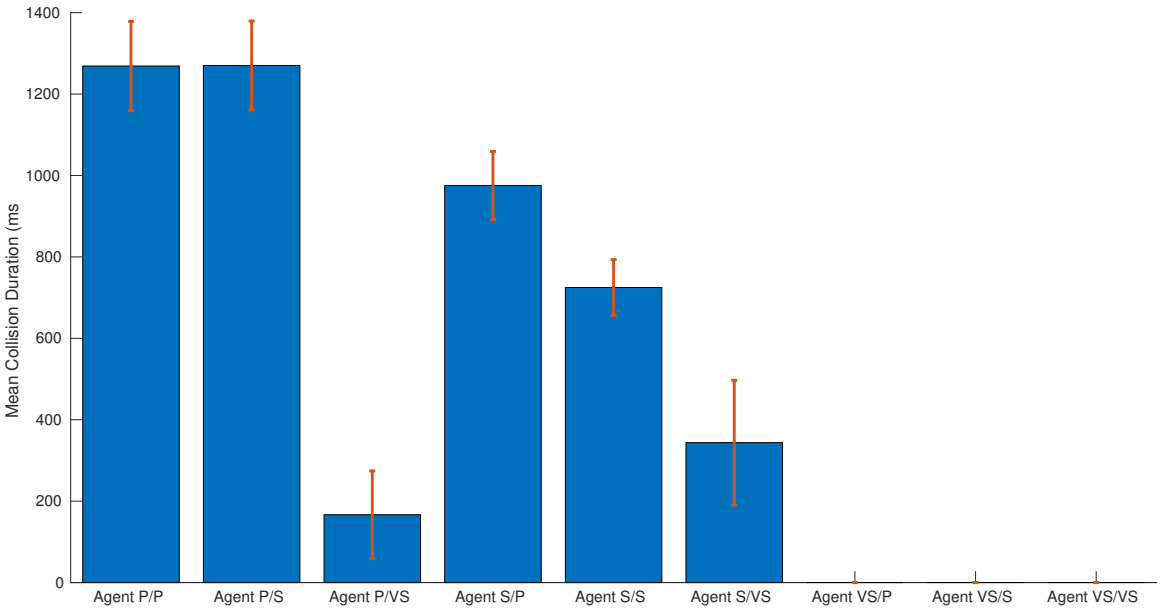

Figure 27. Bar chart depicting mean collision duration per run on the barrel race task for each agent.

As with the navigation tasks above, agent time on task is well modeled by a normal distribution as indicated by a chi-squared test for normality. Since obstacles were spread out and there are no dynamic obstacles, there were fewer factors influencing agent times. Thus, the variance of these agents was typically much less than in the navigation tasks. Tables 17 and 18 present a statistical analysis of agent mean time on task and collision duration respectively for the barrel race task.

Table 17. Statistical summary of mean completion time on the barrel race task.

Agent	Trials	Mean (s)	CI 95%	Std. Dev. (s)
S/S	85	28.09	[27.87, 28.32]	1.03
S/P	93	27.99	[27.78, 28.20]	1.02
S/VS	91	35.58	[34.06, 37.09]	7.28
P/S	96	27.80	[27.60, 28.00]	0.99
P/P	93	27.84	[27.66, 28.03]	0.91
P/VS	80	36.98	[35.59, 38.37]	6.24
VS/S	112	20.45	[20.43, 20.47]	0.13
VS/P	119	20.38	[20.36, 20.41]	0.13
VS/VS	107	25.76	[25.72, 25.81]	0.24

Table 18. Statistical summary of mean collision duration on the barrel race task.

Agent	Trials	Mean (ms)	CI 95%	Std. Dev. (ms)
S/S	85	724	[656, 793]	318
S/P	93	975	[892, 1059]	406
S/VS	91	344	[191, 497]	736
P/S	96	1270	[1161, 1379]	539
P/P	93	1269	[1159, 1379]	533
P/VS	80	167	[59, 274]	484
VS/S	112	0	[0,0]	0
VS/P	119	0	[0,0]	0
VS/VS	107	0	[0,0]	0

These results indicate that Agent VS/P and Agent VS/S, consisting of a combination of vector summation and priority-based paradigms, to be the fittest agents in the cohort with significantly faster times and no collisions. Interestingly, Agents P/VS and S/VS, the "inverse" Agents of VS/P and VS/S, were the least fit. The relationship between "inverse" agents is an interesting result. The P/VS and S/VS agents were slowed considerably as they approached the barrels by erratic, oscillatory movement toward and away from the barrel caused by the higher priority-based arbiter switching between a behavior that pulled the agent closer (`gotoXY`) and one that pushed the agent away (`avoid`). By contrast, the VS/P and VS/S Agents experience a smoother transition between the `gotoXY` and `avoid` behaviors due to the vector summation arbiter. These results demonstrate that arbiter logic and organization both influence agent performance.

4.1.4 Discussion and Summary.

In general, the vector summation, or potential field methodology, based agents performed safely, i.e., did not exhibit erratic movement which could cause loss of vehicle control, and were able to complete the assigned tasks. These agents flew smoothly, generated fewer hits, and still flew relatively fast. This result was expected since potential field approaches have long been shown to be effective in simple, local navigation tasks. The fact that a combination of paradigms proved most fit was an unexpected conclusion.

To illustrate this effect, Figure 28 depicts the typical path of Agent VS/S, a mixed paradigm agent, versus Agent VS/VS, a pure paradigm agent.

The "pure" vector agent runs the course very smoothly at the expense of being overly conservative. The mixed agent followed a more direct path to the next barrel and remained close to barrels during a turn because only the `Orbit` behaviors were active. These results seem to imply that no single reactive paradigm, embodied by the S/S, P/P, and VS/VS Agents, is dominant, even for a relatively simple, navigation-based task.

Based on these results, the primary experimental objective, which was to determine which reactive robotic paradigm produces the fittest agents on the navigation-based tasks, was satisfied. For these tasks, a specific combination of vector summation and priority based paradigms performed best on the barrel race task. The results for the navigation tasks were less clear, but indicated that some priority based approaches are faster and less safe and potential field approaches are slower and more safe. While a combination of paradigms may not always be optimal, this experiment demonstrates that some tasks do benefit from flexibility in arbiter logic and organization. Since the UBF supports many arbiter types and arrangements it follows that it is an improvement over traditional reactive paradigms both in terms of flexibility and performance

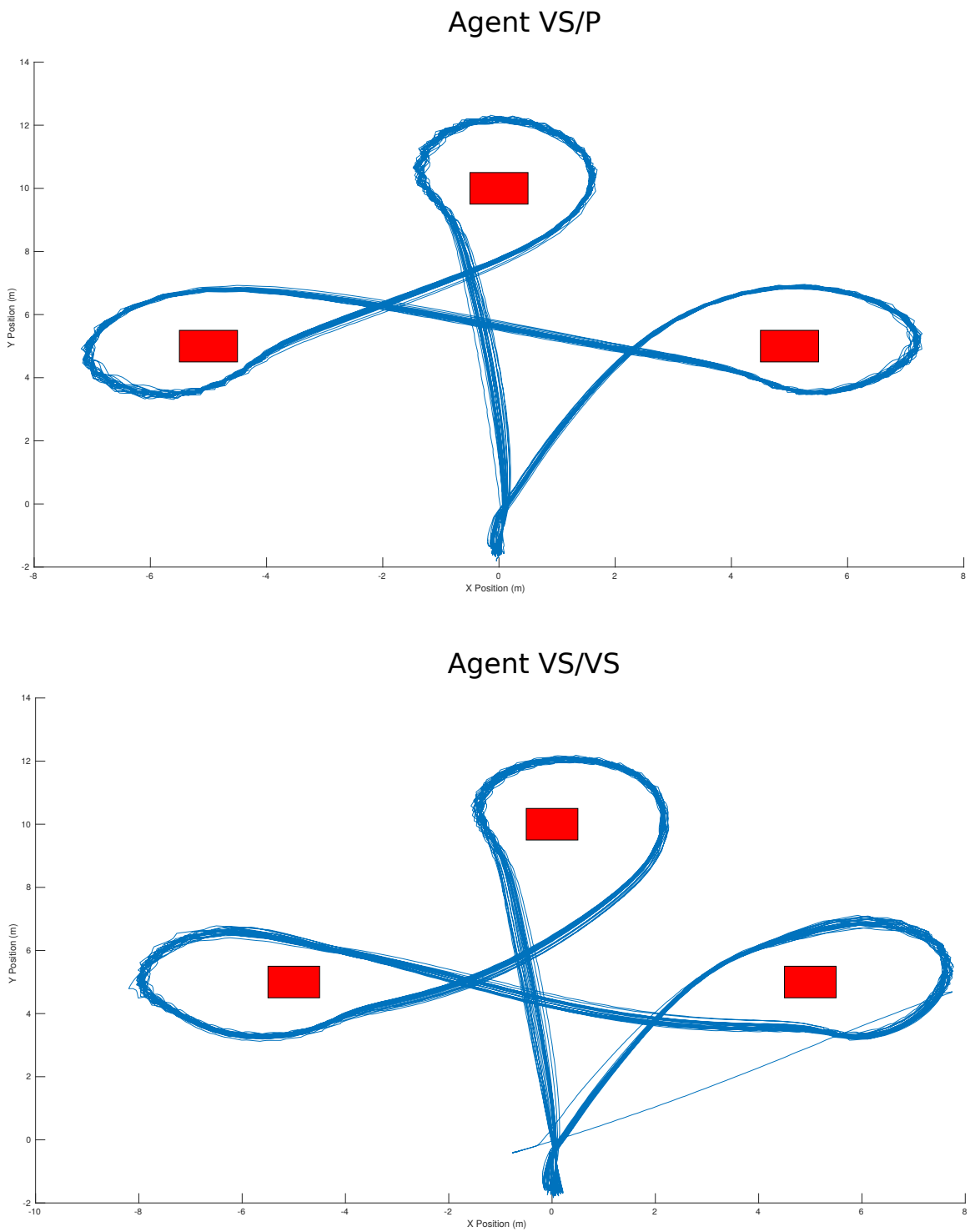

Figure 28. Comparison of Agent VS/P trajectory (top) with Agent VS/VS (bottom) over 25 runs.

in some behaviors. Furthermore, all agents were capable of completing the given tasks indicating that the implementation itself is viable for simulated agents.

4.2 Experiment 2: Flight Test Validation of the Behavior-Flexible UAS Development Platform

The following sections present the results of the second experiment of this research. The experiment consisted of seven tests flights which established the regulatory compliance of the platform (IQ4) and compared the performance of simulated and physical UBF agents (IQ5). The following analysis is intended to analyze variation in agent performance between the two domains by characterizing the flight controllers of each and then comparing the agent performance on the navigation-based task. The section begins with an analysis of controller performance and control loop rate statistics of the `tf_agent`.

4.2.1 Controller Performance Comparison Results.

Tables 19 and 20 summarize the performance characteristics of each controller.

Table 19. Controller vertical motion performance statistics comparison between an emulated and real controller. Percent difference is calculated with simulated results as a baseline.

	Simulated	Actual	Difference
RMSE Position Hold (m)	0.01	0.06	500.0%
RMSE Path Following (m)	0.04	0.28	600.0%
Rise Time (s)	2.47	3.80	53.8%
Latency (s)	0.25	0.52	116.0%
Frequency Cutoff (rad/s)	0.68	0.59	-12.3%

The real controller experienced significantly more positional error, as measured by the root-mean-square-error (RMSE) between the desired setpoint and actual position, in both the vertical and horizontal plane. Both controllers exhibited an increase in RMSE from the `offboard_example` position holding task to the `circle_agent` path

Table 20. Controller horizontal motion performance statistics comparison between an emulated and real controller. Percent difference is calculated with simulated results as a baseline.

	Simulated	Actual	Difference
RMSE Position Hold (m)	0.05	0.34	629.0%
RMSE Path Following (m)	0.12	0.87	621%
Rise Time (s)	2.82	2.21	-21.6%
Latency (s)	0.66	1.12	69.7%
Frequency Cutoff (rad/s)	0.71	1.09	-54.2%

following task. Interestingly, the percent difference of RMSE between position hold and path following tasks stayed relatively constant. This result suggests that position control loop tuning parameters, which determine how responsive the multirotor is to error between actual position and desired setpoint, did not contribute significantly to variation in overall agent performance. If these tuning parameters had contributed, one would expect a larger difference between the static and dynamic cases since, in the static case, error due to a sluggish response would average out as the vehicle remained at the desired setpoint. In the dynamic case the error would accumulate. Thus, variation in controller positioning error from simulation to reality is likely due to a combination of differences in the tuning parameters of the attitude controller, airframe characteristics, and simulation fidelity.

In general, the simulated controller was slightly more responsive, with a faster rise time and lower latency compared to the actual controller. The variation in system response between the domains is likely the result of simulation inaccuracies and tuning parameter differences. Overall, observed controller performance differences are small and contributed little to variance in agent performance between simulation and flight testing.

One controller performance aspect which significantly impact positional accuracy was the stability of the Local Position Estimator (`LPE`) estimator. Position estimates, delayed by networking latency, were occasionally dropped by the estimator since

they were over 0.2 seconds old. During dropouts, position estimates rapidly diverged causing the aircraft to momentarily jerk. A few dozen timeout events would occur during each flight with events lasting around a 0.5 to 1.0 second. Since flight testing, the PX4 development team have chosen to deprecate the estimator, in favor of the Extended Kalman Filter Version Two (`EKF2`) estimator, because of these stability problems.

Overall, variation in positional error between simulation and reality had the greatest contribution to overall variation in agent performance. The first step in reducing this variation would be to switch the platform to the `EKF2` estimator, which would increase the stability of local position estimates, and decrease networking latency, which would reduce the number of dropped position estimates from the motion capture system. Second, finer tuning of flight controller control loop parameters might yield slightly better performance. For these flight tests, relatively low controller gain values were used to increase stability at the cost of reducing the reactiveness of the flight controller. Notwithstanding these potential performance increases, a certain level of variation between simulation and reality is unavoidable without more performant control strategies for the vehicle or higher fidelity simulation.

Table 21 summarizes the control loop rate statistics between simulated and actual controllers.

Table 21. Controller rate statistics comparison between an emulated and real controller. Percent difference is calculated using simulated results as a baseline.

	Simulated	**Actual**	**Difference**
Samples	27,406	37,638	N/A
Mean Loop Rate (Hz)	100.37	100.00	-0.37%
Loop Rate SE (Hz)	0.65	0.54	-0.20%
Min Loop Rate (Hz)	46.20	90.38	96.63%
Max Loop Rate (Hz)	120.78	111.39	-7.77%

For non real-time systems, both controllers maintained very consistent loop rates.

Both systems achieved a mean loop rate within 0.4 Hz of the target loop rate of 100 Hz with standard deviations less than one hertz. Interestingly, the variance for the simulated controller tended to be higher than that of hardware. This is most likely due to the PX4 being emulated on a non real-time system versus being run on real-time hardware. Although the complexity of behavioral logic certainly has an effect on the ability of the onboard processor to meet the target loop rate, no noticeable difference in mean controller loop rate was observed across different behavior sets. As more computationally intensive nodes (such as computer vision and planning algorithms) are added to the base ROS system, the responsiveness and stability of the controller could be impacted negatively

4.2.2 Agent Performance Comparison Results.

The following section addresses qualitative and quantitative performance differences between simulated and real UBF agents. Three agents were flown during flight testing: Agent VS, Agent VS/P, and Agent VS/VS. Since safety was a primary concern, these agents were selected for the smooth, predictable motion demonstrated in simulation. Figures 29, 30, and 31 depict observed agent trajectory in the various scenarios.

Figure 29 demonstrates that Agent VS approaches the obstacles in the same manner in both simulation and reality. Both agents extend out, in positive X, to approximately five meters and up to approximately six meters in positive Y. This result suggests that for a static course flown at slow speeds, simulated results may have high predictive capability for real world-behavior. Figure 30 highlights a problem with the generalizability of simulated results however. In the simulated run, the agent was unable to complete the course due to getting caught in a local minimum. The actual agent was able to complete the run, likely due to the random motion of

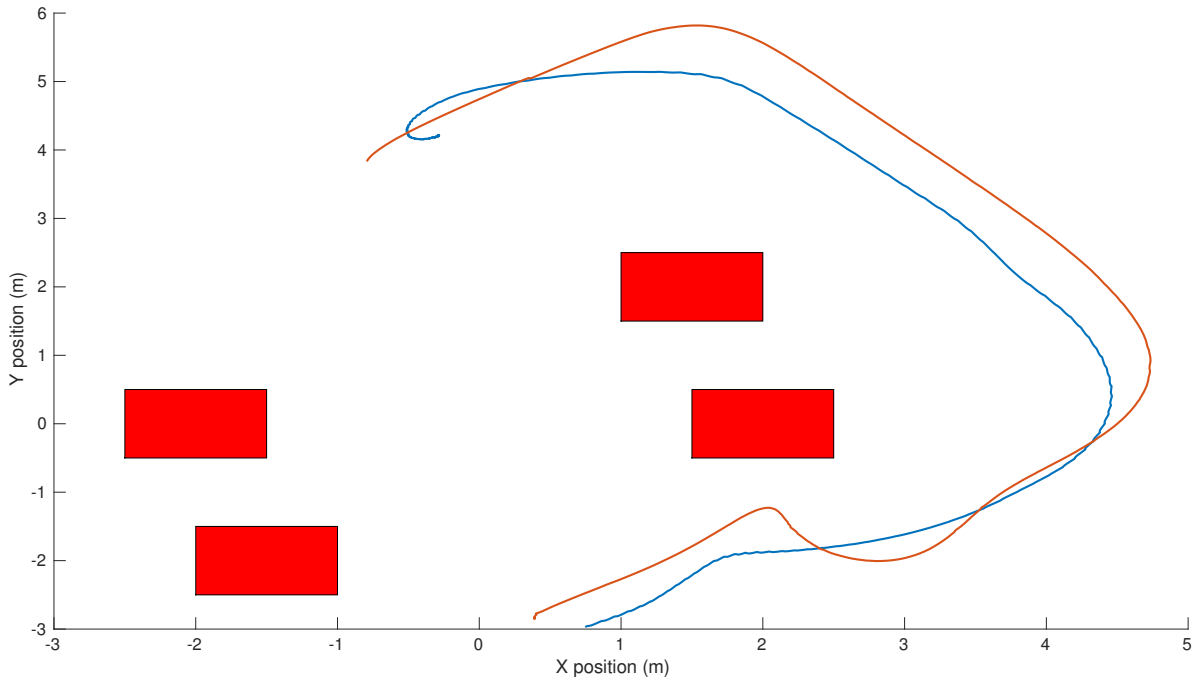

Figure 29. Comparison of simulated Agent VS trajectory (blue) with actual trajectory (orange) on the static navigation task.

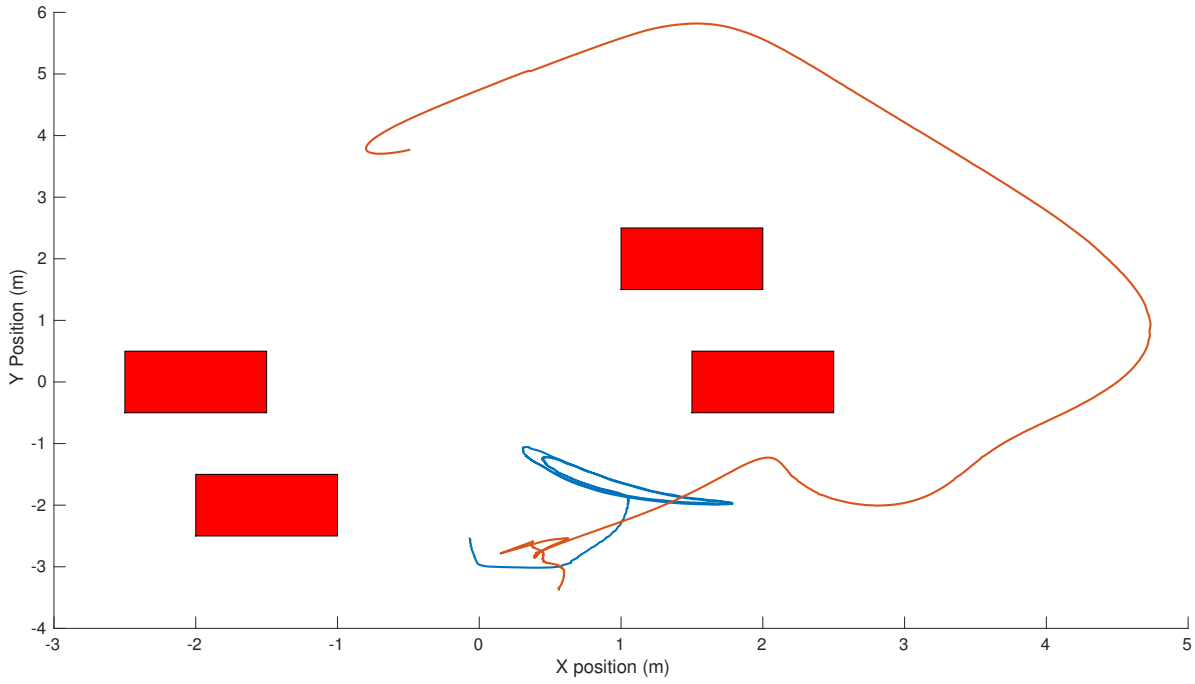

Figure 30. Comparison of simulated Agent VS trajectory (blue) with actual trajectory (orange) on the dynamic navigation task.

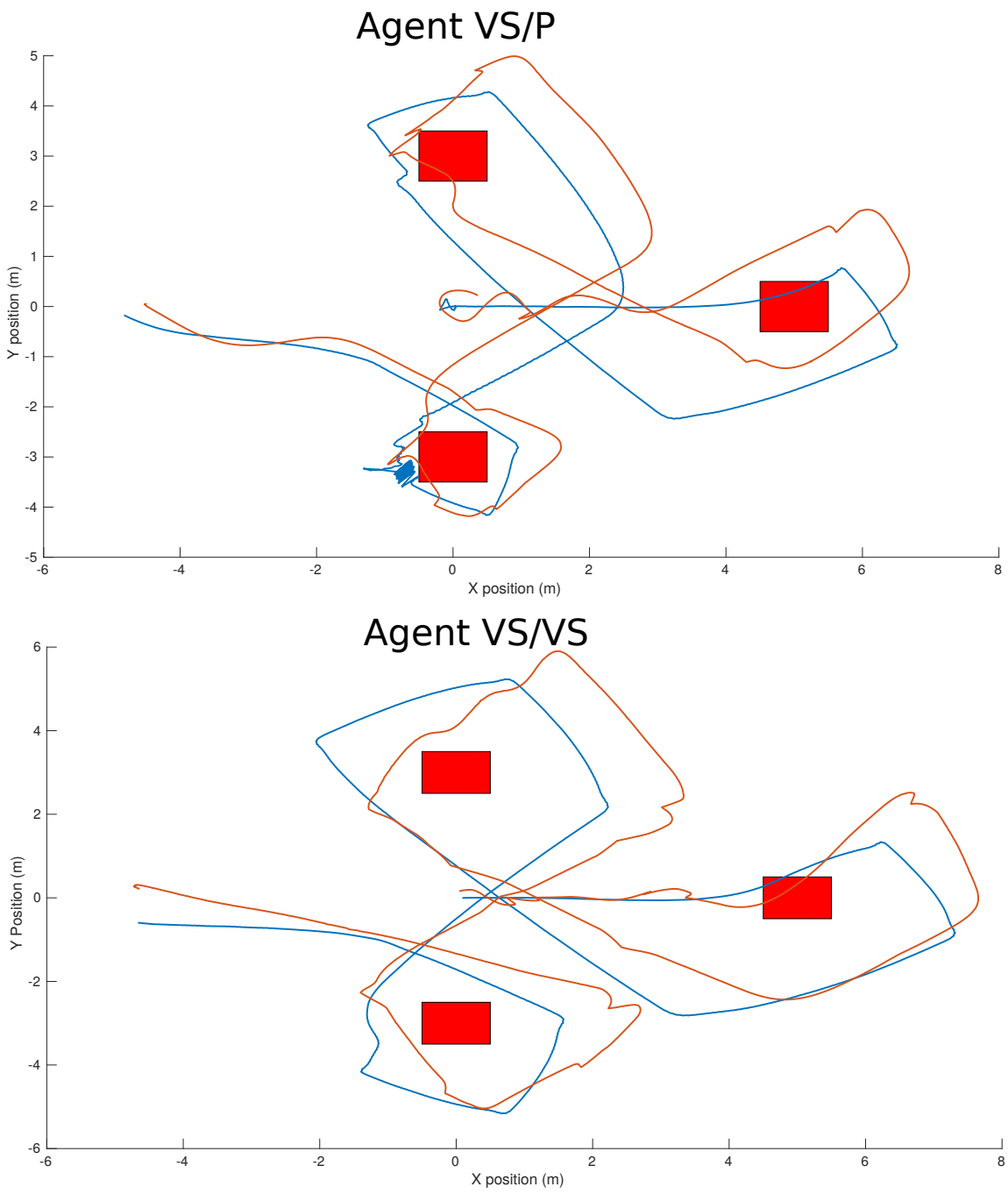

Figure 31. Comparison of Agent VS/P and Agent VS/VS trajectory in simulation (blue) and during flight test (orange) on the barrel race task.

the vehicle"dislodging" the agent from the local minimum near (2,-2). Initially, both agents seem to proceed to this minimum before diverging.

Figure 31 is significant because it demonstrates the performance of two different agents on the same task. Differing arbiter logic clearly had an effect on agent performance in both simulation and flight test. Agent VS/P stayed much closer to barrels than Agent VS/VS just as predicted by simulation. In terms of the agent trajectory, the simulated behavior appears to agree with observed performance.

Tables 22 and 23 present the quantitative performance metrics for this experiment.

Table 22. Comparison of real and simulated agent mean completion time on several navigation tasks. Percent difference is calculated using simulated results as a baseline.

Task - Agent	Simulated	Actual	Difference
Barrel race - Agent VS/VS (s)	105.12	63.84	-35.5%
Barrel race - Agent VS/P (s)	125.8	76.61	-39.1%
Dynamic navigation - Agent VS (s)	DNF	19.39	-
Static navigation - Agent VS (s)	22.29	34.50	55.8%

Table 23. Comparison of real and simulated agent mean collision duration on several navigation tasks. Percent difference is calculated using simulated results as a baseline.

Task - Agent	Simulated	Actual	Difference
Barrel race - Agent VS/VS (ms)	264	627	137.5%
Barrel race - Agent VS/P (ms)	891	429	-51.9%
Dynamic navigation - Agent VS (ms)	0	0	0%
Static navigation - Agent VS (ms)	0	891	-

The quantitative agent performance metrics in this experiment, e.g. time on task and collision duration, show considerable variance in quantitative agent performance across domains. Additionally, there is no apparent trend since the real agents were faster in some tasks and slower in others and incurred more collision duration in some tasks and not in others. The cause of this variance is not clear and is likely due to a myriad of confounding factors between simulation and flight test. The first, and likely largest factor, is the high positional error of the real controller. The high positional

error, coupled with estimator instability, contributed to the "noisy" path that real agents took. This noisy path was, at times, beneficial to real agents. In figure 31 simulated Agent VS/P nearly gets trapped in a local minimum near the first barrel whereas the real agent did not. Another potentially significant factor is simulator fidelity. The multirotor dynamics model used in the simulator is only approximate which might explain the differing trajectories observed.

4.2.3 Discussion and Summary.

The objectives for this experiment were three-fold. The first was to validate that the platform is safe, stable, and compliant with testing regulations. During the flight test experiment, seven successful flight tests were conducted. With the help of a safety pilot, the multirotor was able to achieve, maintain, and terminate flight safely. During flights, the safety pilot was able to transfer control to an from the onboard agent in a reliable and timely fashion. Additionally, a reliable GCS connection was maintained throughout each flight. Although the platform is not yet mature, its compliance in these safety critical requirements will allow future testing of the system.

The second objective was to determine the degree to which simulated agent performance generalizes to real-world performance. In terms of qualitative performance, agent behavior was relatively similar across domains as evidenced by the similarity of agent trajectory during tasks. While the trajectories observed in flight tests were noisier than simulated results, the general shape of the path taken by the agents were basically the same. In terms of qualitative performance, however, there was much more variance between agent domains.

The causes for this variance are likely multivariate and warrant further study. Possible sources of variance are fidelity of the simulation, the fidelity of PX4 emulation, controller tuning parameters, differences between the vehicles, and the estimator sta-

bility. The objective was met since this experiment showed that simulation is useful for making coarse-grained predictions, such as the general path the agent will take, but starts to break down for finer-grained predictions like completion time and the precise path the agent will follow.

The third objective was to compare controller performance between the domains. Both sets of loop rates were remarkably stable with regard to their target rate of 100 Hz. Actual hardware slightly outperformed simulation due to the real-time capability of the Pixhawk. In terms of performance, the simulated controller held position in the vertical and horizontal planes extremely well compared to the actual controller. The simulated controller was slightly more responsive than the actual hardware, but only by fractions of a second. These results indicate that differences between the emulated and actual controller contributed much of the observed variance in agent behavior between the domains.

Finally, and most importantly, the results of this series of flight tests show that the hypothesis is true. The UBF implementation in ROS was demonstrated to be a viable, behavior-flexible platform for physical multirotors. This experiment carries forward the results of the first experiment to the real-world, showing that real-world agents are capable of executing dynamic, unstructured tasks. Furthermore, while simulations are not perfect reflections of reality, they are similar enough that useful predictions of real world agents can be made on the basis of simulated behavior.

V. Conclusion

The objective of this research was to assess the feasibility of creating a behavior-flexible UAS development platform from open-source software components. To address this research question, the UBF was integrated with a state-of-the-art robotic software framework and open-source flight controller. The results show that agents running on this platform are largely predictable on the basis of their simulated results, responsive to the environment based on the observed controller performance, and capable of competently executing tasks as evidenced by agent performance on the navigation task set. This thesis also documents the first use of a UBF agent aboard a UAS. While the technological readiness of the platform is still in its infancy, it serves as a foundation for further refinement and modification. This chapter recounts the investigative questions addressed in this work along with their contributions to future research and the DoD. The chapter concludes with recommendations for future work and final remarks.

5.1 Summary

The first investigative question of this research established the current state-of-the-art with regard to RSFs and open-source flight controllers. Since multirotor UAS were the target vehicle type, a review of multirotor control strategies and a survey of open-source flight controllers was conducted. The multirotor control problem is difficult but capable, open-source controllers are currently under active development. The research elected to use the Pixhawk flight controller running the PX4 firmware due to its modular codebase and interoperability with ROS. ROS connects the Pixhawk controller to the UBF behavior logic. The decision to use ROS as the RSF of choice was made after an comprehensive survey of RSF projects. ROS represents one of the

most mature and well supported RSFs currently available. ROS provides a capable middleware for a large number of components and algorithms that are free to use and already proven. Additionally, the debugging, logging, and introspection tools provided by ROS were an invaluable resource throughout the research process.

The second investigative question of this research proposed a design for a behavior-flexible development platform which integrated the UBF with the aforementioned software components. The inclusion of ROS increased the adaptability of the system since the addition of sensors, motors, and algorithms are facilitated by the publish and subscribe middleware. The responsiveness of the system was enhanced by the inclusion of the Pixhawk flight controller since the flight controller software is executed separately from the onboard computer on a real-time system, i.e. the Pixhawk. The distributed nature of the system also increased the safety of the platform since the safety critical code is run on a real-time system with a backup processor. The UBF improved the behavioral-flexibility of the system by allowing novel combinations of reactive robotic paradigms and reuse of behaviors. These features were shown in simulation to improve agent fitness on navigation-based tasks over traditional reactive robotic implementations.

The third investigative question identified which reactive paradigms performed best on a set of navigation-based tasks. In the static and dynamic obstacle navigation tasks, the vector summation (V/S) arbiter experienced the shortest collision duration and the priority (P) arbiter was fastest. In the barrel race task, the vector summation with priority or subsumption arbiters (VS/P and VS/S) agents performed better than any pure combination (e.g. a P/P, VS/VS, S/S agent). This result is significant for two reasons. (1) The flexibility offered by the UBF in the selection, use, and organization of arbiter logic is an improvement over traditional approaches for some tasks and (2) potential field methodologies facilitated by the vector summation arbiter

produce competent UAS agents executing navigation-based tasks.

The fourth investigative question established the safety, stability, and regulatory compliance for the platform through flight test. On seven flight tests, the UAS was able to achieve, maintain, and terminate flight safely. The GCS connection was stable during flights and the safety pilot was able to transfer control. Establishing this basic compliance will expedite future testing at AFIT since a test review board (TRB) and safety review board (SRB) will be able to reference this historical performance data for the platform.

The final investigative question determined the degree to which simulated agent behavior predicts actual agent behavior. For course-grained behavior, such as the approximate trajectory the agent will take, simulation provides useful predictions of real agent behavior. For fine-grained performance metrics, such a collision duration and time to complete a task, simulation may not be as useful. A characterization of controller performance identified a disparity in positional accuracy for simulated flight controllers and real flight controller to be a significant source of error. Although simulations do not perfectly predict real-world behavior, they are still a useful tool for determining course-grained agent behavior. Accurate simulation is a requirement for an autonomous development platform to ensure that agent logic is safe and correct before risking hardware and time in flight tests.

Since each investigative question was answered in the affirmative, the hypothesis was shown to be true and the research question was answered; It is possible to develop a behavior-flexible development platform for UAS agents using open-source software components by integrating the UBF with an RSF. The platform presented in this work provides a behavior-flexible platform for future research efforts involving autonomous UAS. The UBF allows researchers to quickly implement, modify, tune, extend, and reuse UAS agents with a variety of behaviors. If additional sensors or ac-

tuators are required, researchers can quickly add them to the system through the ROS middleware. The platform is also portable to different vehicle types through the PX4 flight controller. Because of these features, future research using the platform can focus more intensely on research interests as opposed to tangential implementation details. In addition to its research benefits, the development platform highlights the benefits of highly modular, behavior-flexible robotic software. Using this approach could significantly lower the cost, shorten the acquisition timeline, and increase the usefulness of autonomous vehicles for the DoD.

5.2 Future Work

This research offers several avenues for potentially fruitful future work. Five areas of future work are suggested below.

- Incorporate a deliberative layer on the platform to implement a hybrid deliberative/reactive framework adding planning capability to the development platform.
- Incorporate a social layer on the platform to allow the agent to communicate with other agents in a multi-agent system adding swarming capability to the development platform.
- Migrate from ROS to ROS2. ROS2 is being developed to support real-time robotic software and a improved, low-latency, high-bandwidth communication middleware called Real-time Publish and Subscribe [47].
- Explore the portability and extensibility of the platform by incorporating additional sensors and testing agents on different vehicle types.
- Explore metaheuristic optimization techniques, e.g. genetic algorithms, to design and tune agents.

5.3 Final Remarks

Future autonomous vehicles will be responsible for an ever growing set of tasks in military, civil, industrial, and exploratory applications. As these systems are entrusted with more complex and important missions, the need predictability, responsiveness, and robustness will grow as well. Thus, the behavioral-flexibility of these systems will become a highly desirable trait, allowing the maximum utility to be extracted from each development effort. Behavioral-flexibility allows designers to refine agent behavior, augment existing systems with additional components, and reuse proven software in new applications. The platform designed, implemented and tested in this work address each of these concerns and while it is immature and imperfect, it represents a small contribution in an increasingly important and relevant field of study. Further refinement of this platform, and other unified platforms like it, offer tremendous flexibility for designers to employ new vehicles, sensors, deliberative and social algorithms, and behaviors without concern for many of the low-level concerns which may have slowed progress in the past.

Bibliography

1. B. G. Woolley and G. L. Peterson, "Unified behavior framework for reactive robot control," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 55, no. 2-3, pp. 155–176, Jul 2009. [Online]. Available: http://link.springer.com/10.1007/s10846-008-9299-1

2. R. C. Leishman, J. Macdonald, R. W. Beard, and T. W. Mclain, "Quadrotors & Accelerometers," *Control Systems Magazine*, pp. 1–51, Feb 2014.

3. R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*. Princeton: Princeton University Press, 2012.

4. "USAF Strategic Master Plan," USAF, 2015.

5. D. S. Board, "Summer Study on Autonomy," USAF, 2016.

6. "China issues guideline on artificial intelligence development," The State Council of the DPRC, Jul 2017.

7. "'Whoever leads in AI will rule the world': Putin to Russian children on Knowledge Day," Russia Today, Sept 2017. [Online]. Available: https://www.rt.com/news/401731-ai-rule-world-putin/

8. D. Tennenhouse, "Autonomous Vehicles: Are You Ready for the New Ride?" MIT Technology Review Insights, 2017. [Online]. Available: https://www.technologyreview.com/s/609450/autonomous-vehicles-are-you-ready-for-the-new-ride/

9. J. L. Hardcastle, "Why Automakers, Tech Giants are Investing Millions in Autonomous Vehicles," 2015, https://www.environmentalleader.com/2015/11/why-automakers-tech-giants-are-investing-millions-in-autonomous-vehicles/.

10. M. Weisgerber, "http://www.govexec.com/feature/slow-and-steady-losing-defense-acquisition-race/," 2015. [Online]. Available: http://www.govexec.com/feature/slow-and-steady-losing-defense-acquisition-race/

11. R. N. Charette, "What's Wrong with Weapons Acquisitions," 2008, https://spectrum.ieee.org/aerospace/military/whats-wrong-with-weapons-acquisitions.

12. M. J. Sullivan, "DOD Is Taking Steps to Address Challenges Faced by Certain Companies," GAO, Tech. Rep. GAO017-644, 2017. [Online]. Available: https://www.gao.gov/assets/690/686012.pdf

13. "LRASM," Jul 2016, https://www.lockheedmartin.com/us/products/LRASM.html.

14. GAO, "Assessments of Selected Weapon Programs." *Report to Congressional Committees*, vol. GAO-15-342, no. March, pp. 1–194, 2015.

15. M. Endsley, "Autonomous Horizons: System Autonomy in the Air Force - A Path to the Future," USAF, p. 27, 2015. [Online]. Available: http://www.af.mil/Portals/1/documents/SECAF/AutonomousHorizons.pdf

16. P. Fitzpatrick, E. Ceseracciu, D. E. Domenichelli, A. Paikan, G. Metta, and L. Natale, "A middle way for robotics middleware," *Journal of Software Engineering for Robotics*, vol. 5, no. September, pp. 42–49, 2014.

17. P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 29–45, 2008.

18. G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet another robot platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 043–048, 2006.

19. "ArduPilot Homepage," 2016, http://ardupilot.org/.

20. L. Meier, D. Honegger, and M. Pollefeys, "PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms," *2015 IEEE International Conference on Robotics and Automation*, pp. 6235–6240, 2015.

21. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," *Icra*, vol. 3, no. Figure 1, p. 5, 2009. [Online]. Available: http://pub1.willowgarage.com/{~}konolige/cs225B/docs/quigley-icra2009-ros.pdf

22. S. S. Lin, "Unified Behavior Framework in an Embedded Robot Controller," Master's thesis, AFIT, 2009.

23. D. Roberson, D. Hodson, G. Peterson, and B. Woolley, "The Unified Behavior Framework for the Simulation of Autonomous Agents," Master's thesis, AFIT, 2015.

24. A. Elkady and T. Sobh, "Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography," *Journal of Robotics*, vol. 2012, p. 15, 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/959013

25. J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, no. 2, pp. 101–132, Jan 2007. [Online]. Available: http://link.springer.com/10.1007/s10514-006-9013-8

26. N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for Robotics: A Survey," in *2008 IEEE Conference on Robotics, Automation and Mechatronics*. IEEE, Sept 2008, pp. 736–742. [Online]. Available: http://ieeexplore.ieee.org/document/4681485/

27. R. R. Murphy, *AI Robotics*. Cambridge: The MIT Press, 2000.

28. N. Michael, D. Mellinger, Q. Lindsey, and V. Kumar, "Experimental evaluation of multirobot aerial control algorithms," pp. 56–65, 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5569026/

29. H. Lim, J. Park, D. Lee, and H. J. Kim, "Build your own quadrotor: Open-source projects on unmanned aerial vehicles," *IEEE Robotics and Automation Magazine*, vol. 19, no. 3, pp. 33–45, 2012.

30. J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," *Machine Intelligence*, vol. 4, no. 463-502, pp. 463–502, 1969.

31. V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*. MIT Press, 1986.

32. R. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986. [Online]. Available: http://ieeexplore.ieee.org/document/1087032/

33. R. C. Arkin, "Motor Schema-Based Mobile Robot Navigation," *The International Journal of Robotics Research*, vol. 8, no. 4, pp. 92–112, 1989.

34. B. G. Woolley, "Unified Behavior Framework for Reactive Robot Control in Real-Time Systems," Master's thesis, AFIT, 2007.

35. E. Gamma, *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995. [Online]. Available: https://books.google.com/books/about/Design_Patterns.html?id=iyIvGGp2550C

36. D. J. Hooper, "A Hybrid Multi-Robot Control Architecture," Master's thesis, AFIT, 2007.

37. J. P. Duffy, "Dynamic Behavior Sequencing in a Hybrid Robot Architecture," Master's thesis, AFIT, 2008.

38. R. Mahony, V. Kumar, and P. Corke, "Multirotor Aerial Vehicles: Modeling, Estimation, and Control of Quadrotor," *IEEE Robotics & Automation Magazine*, vol. 19, no. 3, pp. 20–32, 2012.

39. A. Tridgell, "ArduPilot and DroneCode," 2016, https://discuss.ardupilot.org/t/ardupilot-and-dronecode/11295.

40. "PX4 Documentation," 2017, dev.px4.io/en.

41. P. Inigo-Blasco, F. Diaz-Del-Rio, M. C. Romero-Ternero, D. Cagigas-Muniz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Robotics and Autonomous Systems*, vol. 60, no. 6, pp. 803–821, Jun 2012. [Online]. Available: http://dx.doi.org/10.1016/j.robot.2012.02.004http://linkinghub.elsevier.com/retrieve/pii/S0921889012000322

42. G. Biggs and B. Macdonald, "A Survey of Robot Programming Systems," *Proceedings of the Australasian conference on robotics and automation*, pp. 1–3, 2003.

43. J. Altmann and F. Gruber, "Using mobile agents in real world: A survey and evaluation of agent platforms," *... for Agents, MAS, and ...*, 2001.

44. N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based RT-system development: OpenRTM-Aist," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5325 LNAI, pp. 87–98, 2008.

45. H. Bruyninckx, "Open robot control software: the OROCOS project," *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3, pp. 2523–2528, 2001.

46. "MAVLINK Common Message Set," http://mavlink.org/messages/common.

47. OMG, "The Real-time Publish-Subscribe (RTPS) Wire Protocol DDS Interoperability Wire Protocol Specification v2.2," 2014. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/09-01-05.pdf

48. "X8 User Manual," 2014, https://3dr.com/wp-content/uploads/2017/03/X8-Operation-Manual-vA.pdf.

49. "Rasberry Pi Homepage," https://www.raspberrypi.org/.

50. "BeagleBone Black Homepage," https://beagleboard.org/black.

51. "Odroid-XU4," http://www.hardkernel.com/main/products/.

52. "Pixhawk Autopilot Documentation," https://pixhawk.org/modules/pixhawk.

53. C. Hendrix, M. J. Veth, and R. W. Carr, "LQG Control Design for a Hovering Micro Air Vehicle using an Optical Tracking System," 2009. [Online]. Available: https://www.afit.edu/ANT/news.cfm?na=detail{\&}ncat=ANT{\&}item=188

54. "ROS Documentation," 2017, http://wiki.ros.org/.

55. "Gazebo," 2017, http://gazebosim.org/.

56. P. Roduit, A. Martinoli, and J. Jacot, "A quantitative method for comparing trajectories of mobile robots using point distribution models," in *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on.* IEEE, 2007, pp. 2441–2448.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2018	Master's Thesis	Sept 2016 — Mar 2018

4. TITLE AND SUBTITLE

Behavior Flexibility for Autonomous Unmanned Aerial Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Bodin, Taylor B., 2d Lt., USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-18-M-011

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally left blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of this work was to assess the feasibility of a behavior-flexible development platform for UAS. This work first establishes the current state-of-the-art with regard autonomy frameworks, robotic software frameworks RSF, and flight controllers for UAS, components necessary for a development platform. The work then proposes a design incorporating the Unified Behavior Framework (a modular, extensible autonomy framework), the Robotic Operating System (an RSF), and PX4 (an open-source flight controller). Using the platform, the work then identifies a combination of autonomous robotic control strategies which are effective for small-scale navigation tasks in simulation. Finally, the work provides a partial validation of the simulated results through flight test. The development platform presented in this work is shown to be robust, responsive, and behavior-flexible both in simulation and reality.

15. SUBJECT TERMS

UBF, Reactive Robotics, UAS, Autonomy

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maj. Jason M. Bindewald, Ph. D., AFIT/ENG
U	U	U	U	119	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636, x4614; jason.bindewald@afit.edu